Development of a Control System and Simulator for a Skid Steer Amphibious Vehicle

Author: Michael F Clarke
Project Supervisor: Mark Neal
April 24, 2010

Submitted in partial fulfilment of the MEng degree at Aberystwyth University

Declaration of Originality

In signing I confirm that:

I understand that there are sever penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.

I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.

I understand and agree to abide by the University's regulations governing these issues.

Michael F Clarke

Acknowledgements

Throughout this project many people have offered their support and help and I wish to thank all of you for your patience, understanding and infallible support during the last few months. Without you, this project would not have been such a success.

In particular I would like to thank Dr. Mark Neal, my project supervisor, for his often insightful gems of wisdom. There were often times when my mind was completely blank and without his influence and guidance in the right direction I fear that this project never would have succeeded.

I would also like to express my gratitude to Tom Blanchard for his invaluable assistance with relation to the hardware aspects of this project and his never ending enthusiasm. I wish him all the best of luck with his dissertation next year now that his industrial year nears the end.

In a similar vain, I would also like to thank Colin Suaze and Barry Thomas for their support during this year.

I would further like to thank Stephen Rooney, Samiwel Thomas, Chris Waters and Michael Jones for their friendship during this period. I would also like to thank my parents. Our conversations over coffee where you relentlessly listened to my complaints, problems, ideas and general 'whitterings' about my project are truly appreciated.

Finally, I would like to thank Michael Jones, Stephen Rooney and Ben Lavery for proof-reading this document. Any mistakes that remain are entirely my fault. I am always one to tinker, and fear that after their final proof reading I may have made many further changes!

To you all, thank you again!

Abstract

With autonomous robotic systems increasingly being used for various tasks including research projects, their ability to operate away from their ideal laboratory environment comes into question. Whilst long term autonomous robotic platforms do exist, few are based on vehicles that provide all terrain maneuverability, and certainly none combine this with skid-steering mechanisms. Inspired by the robotic requirements of the Rees Scan project in New Zealand, this dissertation aims to show the development of an all terrain robotic platform based on an ARGO 6x6 Amphibious Petrol Powered Skid-Steer Vehicle. The design and development of a control system, including software and hardware, is discussed. Further, a simplistic simulation environment is developed in an attempt to model the performance of the robotic platform on different surfaces. It is shown that the control of such a vehicle is possible with a reasonably high level of precision with relatively low cost components, suggesting that with more expensive hardware or different techniques, high-precision control would be possible. Further, it is shown that based on the various characteristics of the robotic platform such as the throttle values, wheel rotations and turn rates it is possible to classify, with some level of accuracy, the varying surfaces allowing for future work towards adaptive control systems able to adjust their performance as the robot moves from one unknown environment to another.

Contents

List of Figures

Chapter 1

Introduction

"I have learnt that our background and circumstances may have influenced who we are, but we are responsible for who we become."

James Rhinehart

1.1 Background

The geography department at Aberystwyth University, in conjunction with various other universities, are in the process of surveying the river bed of the Rees River in New Zealand. They have various research goals of which one in particular specifically relates to this project.

> "To use Terrestrial Laser Scanner (TLS) to derive hyperscale topographic models of the Rees River before and after floods in spring-autumn 2009-2010." [14]

In order to accomplish their research the team are performing surveys of the river bed every time it floods during a period of one year. The resulting data is then being analysed to identify changes to the river bed, which will hopefully provide answers and conclusions to their questions and theories.

Traditionally the measurements and readings for this kind of survey are taken using a manual process [4]. However, the computer science department at Aberystwyth University have helped to semi-automate the process by attaching various sensing and scanning equipment including Global Positioning System (GPS), Accelerometers and Terrestrial Laser Scanner (TLS) to a ARGO 6x6 Frontier (ARGO) based amphibious vehicle. The ARGO is currently driven and operated manually.

This project aims to further the work already completed towards automation by introducing appropriate mechanical, electronic and software components such that the ARGO can be autonomously driven by a computer control system.

In addition to the control software the project aims to produce a simulation environment to allow testing and development of code for eventual use on the ARGO in the hope that this will allow future software to be developed in a laboratory environment before being deployed onto expensive physical hardware, reducing the possible cost of mistakes, and hopefully decreasing the time required for development and testing of new software.

It is further anticipated the this project will significantly help towards the goal of producing a fully autonomous robotic platform that could be used in various other areas of research currently being undertaken at Aberystwyth University such as the scanning and surveying of possible archaeological sites or the 'collaboration' of varying autonomous robotic systems.

1.2 ARGO 6x6 Frontier Vehicle

Figure 1.1: The ARGO 6x6 Frontier Amphibious Skid-Steer Off-Road Vehicle

The vehicle currently being used for the research in New Zealand is an ARGO 6x6 Frontier (Figure 1.1). The ARGO was never designed to be a robotic system, but as a off-road amphibious vehicle. This means that there are various aspects of the ARGO which could be considered unusual in a robotic system. Some of these are discussed in this section.

1.2.1 Amphibious

The vehicle is amphibious, and as such is capable of driving both on land and in water. Further, it can do this without any modifications, adjustments or configuration changes - the vehicle can drive directly from land and into water, continuing operations as normal.

However, in water it is clear that the vehicle will perform differently than on land. According to the manufacturers website[1] the vehicle can achieve a top speed of 3.5 mph (5 km per hour) in water and up to 22 mph (35 km per hour) on dry land.

This clearly has implications for the development of a control system. It is potentially the case that in order for the vehicle to behave appropriately the control system will need to be able to compensate and adapt to these differing conditions as they occur.

1.2.2 Steering

The vehicle uses a method known as skid-steering. This effectively involves locking one set of wheels (via applying the breaks on the left or right side of the ARGO) and adjusting the throttle to a point where the vehicle skids. This again has the potential to present various challenges for automation.

The ground surface is expected to have a considerable impact. For example, in wet grassy conditions it is expected that the vehicle will skid with less effort than on dry concreted surfaces. This will clearly mean that the control software will have to adjust both the

[1] ARGO manufacturers website: `http://www.argoatv.com`, Last accessed 13th October 2009.

steering column and throttle at the same time, and in the correct proportions, for the right amount of time, dependant on the surface conditions, to achieve a desired turn angle.

Further, skidding is clearly not going to be as accurate as a 'conventional steering system' found on many purpose-built robotic systems and automobiles. As such it is possible that after the control system has cut the throttle and unlocked the wheels the vehicle could conceivably continue to skid. In such an eventuality the control software would have to detect and compensate for this, and preferably have a method of preventing this in the first place.

It may be possible to perform surface type identification by gathering enough sensor readings during turns and maneuvers. If this is the case it may further be possible to build characteristic profiles of performance on the various different surfaces in order to better tune the control system, and to enable more accurate representations of movements in the simulator environment.

1.2.3 Petroleum Internal Combustion Engine

The ARGO uses a mechanical internal combustion engine fulled by petroleum. Such engines have considerably more mechanical complexity than an electrical system with a motor and battery. As such this presents various challenges, and also some opportunities.

One clear problem will be that the vehicle will periodically need re-fueling as more conventional options such as solar power will not be possible. This will greatly impact the potential for long periods of autonomy. In addition there will almost certainly be timing issues with the engine that will need to be monitored and accounted for. One possible example of this is the time between the increase in throttle and the resulting acceleration. There will almost certainly be a time lag and this could potentially add further complications for the steering.

However, there are some advantages that should also be noted. Having a petrol engine effectively means an "unlimited" amount of electrical power is available for computer equipment while there is sufficient petrol. In addition, the various mechanical components (such as wheel drive chains) could pose possible targets for sensors to be able to measure speed without necessarily requiring any mechanical modifications to the wheel axles or the engine itself.

1.3 Current Control System Techniques

There are a number of techniques already in wide use for the development of robotic control systems. Those which may be of value for this project are described in this section.

1.3.1 Open and Closed Systems

There are two classifications of control systems, those which use feedback are called 'closed systems'. They monitor the state of a system and perform actions to ensure that the state is maintained, or that a desired new state is achieved. A primary example of closed loop systems includes Proportional Integral Derivative (PID) controllers discussed later in this section.

Open loop systems work by using a model of the system to decide on actions to perform. Such systems are only useful when the responses of the system can be accurately modeled and can be guaranteed. If this is not the case then a open loop control system could, for

example, attempt to turn a motor on at a specific speed by delivering a certain voltage and current. However, when under load the motor may require more power, and so the motor may not achieve the desired speed. A open loop control system would not be able to compensate for the extra power requirements as it has no feedback informing it that the motor is running slower than required.

1.3.2 PID Controllers

PID controllers [9] are used in many systems as a method of closed-loop control. In their simplest form they use the process variable, pv, from a system being controlled (that is the current state of the system) and calculate the error for a point in time, $e(t)$, against the set point, sp, (the state the system should be achieving), as shown in Equation 1.1. This error is then used to continually calculate appropriate values to move servos, actuators, motors or other mechanical components so that the desired set point can be achieved.

$$e(t) = sp - pv \tag{1.1}$$

PID controllers are built from three separate components. P or the proportional component (Equation 1.2) is used to control the system such that it smoothly reaches the set point by adjusting the speed of the system proportionally to the current error. I or the integral (Equation 1.3) is used to prevent steady-state-errors which can sometimes occur when using P controller. If the error is not decreasing I will continue to increase (or decrease) effectively overriding P until the system responds. However, this can lead to over-shoot of the set point and as such D or derivative control (Equation 1.4) can be added to compensate.

$$P = K_p e(t) \tag{1.2}$$

$$I = K_i \int_0^t e(\tau) d\tau \tag{1.3}$$

$$D = K_d \frac{d}{dt} e(t) \tag{1.4}$$

Although the three values are often used together to form a PID controller as shown in Equation 1.5, the three values of P, I and D can be used on their own or in any combination. It may be the case that a simple P or PI controller provides sufficient control that the use of D is not necessary.

$$MV(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \tag{1.5}$$

1.3.3 Fuzzy Logic

Fuzzy Logic is a method of making definite decisions based on imprecise, noisy or only partially accurate data. The general idea is that we as humans often make decisions with partial or unclear information, and we are usually right. We can make statements about the world such as "the temperature is hotter today than yesterday" without having precise numerical measurements.

Such systems make decisions by evaluating a collection of varying system states. For example, if attempting to classify the height of an object, fuzzy logic may have three fuzzy scales: 'object is tall', 'object is small', 'object is of medium height'. Each scale will have a value of 1 to 0. As such, a tall object may result in a value of '1' in the 'object is tall' state, whilst values of '0' may be used for the states 'object is small' and 'object is of medium height'. Further, with fuzzy logic, states can be represented such as 'the object is not of small height (0), it is slightly tall (0.1), but is very medium (0.9)'. This would then be evaluated to suggest the object is in the boundary between medium and tall and as such appropriate control operations could be performed for an object of that approximate height.

Fuzzy Logic can therefore be used in closed-loop control systems as a way of making control decisions based on sensor data that may be noisy, imprecise, or where the data input is only partially available.

It has been used with varying success [20, 16] in the development of control systems for robotic sailing boats and for the automation of steering vehicles [7] amongst other applications.

1.3.4 Deliberative, Reactive and Hybrid Approaches

As well as the algorithmic methods already discussed such as PID controllers, there are also different approaches to the control of systems in terms of the amount of processing that should take place to make decisions and where that processing should execute. This has lead to three categories of control systems:

- **Deliberative** control systems are those where every control decision takes place within a central processing unit. Data from sensors is captured, analysed and a decision on how to move servos, actuators and motors is made based on a set program or algorithm. It could be considered analogous to humans taking sensor input from eyes and ears and processing them in the brain before sending control decisions via the nervous system to arms, legs etc. Such control systems often require a model of the real world in order to make sensible decisions. They tend to have relatively deterministic responses to different stimuli due to the implementation of the main control algorithms.

- **Reactive** control systems are usually built from individual small control systems or agents which culminate into an overall robotic system. In general, each component of the control system make their own decisions regarding sensor input, possibly notifying other agents of their discoveries. In such control systems the reactions of any individual component does not necessarily rely on the input from others. For example, if a robotic system had a manipulator, all the appendages may operate individually. If a sensor on one appendage detects that it is applying too much pressure it may reduce the pressure whilst the remainder of the appendages would continue to operate as normal. Such control systems can often be related to impulse reactions seen in living animals. Due to the complexity of such systems it is often the case that the behaviour is emergent and perhaps less predicable than deliberative control systems. However, the response times can often be improved using such systems.

- **Hybrid** control systems are those which use a combination of both reactive and deliberative control methods. In such systems the deliberative system may be programmed to achieve a specific goal, whilst the reactive system may be in place to respond to

changes in the environment when the deliberate systems would fail. Using the analogy of a human once again, this is clearly how we operate. Our brain provides high-level targets such as move arm from point a to b, perhaps crossing an unanticipated hot surface, whilst our instinctive nature prevents us from burning ourselves by reacting to the sudden change of temperature.

Chapter 2

Proposed System

"It is the mark of an educated mind to be able to entertain a thought without accepting it."

Aristotle

2.1 Overview

As already discussed this project primarily involves the development of a control system for an ARGO amphibious vehicle and also a simulator environment that, to a reasonable degree of accuracy, simulates the performance of the control system and the movements of the ARGO in differing conditions. This results in a number of constraints and requirements for the resulting system. Ultimately it is proposed that the final system should be capable of controlling the movements of the ARGO. This is likely to include features such as steering control and throttle control.

It should be made clear that this project is a research project rather than commercial and therefore in addition to the software engineering problems there are a number of research questions or goals which are detailed later in this chapter.

This chapter details further information about the software engineering and research goals along with any assumptions that will be made regarding the software, vehicle and the operating environment.

2.2 Research Objectives

Whilst robotic platforms are widespread, after significant investigation, it has become clear to the author that the automation of an ARGO (or *sufficiently* similar) vehicle has never been attempted before. This has lead to a number of questions both about the automation of such a vehicle and the effects of its surroundings:

1. Can a vehicle such as an ARGO be automated, and what complexity of both software and electronic components are required?
2. What level of accuracy can be expected from a skid-steering system, and to what extent is a control system more or less accurate than a human operator?
3. What effects do surface type have on the operation of the vehicle?
4. Given observed characteristics of the vehicle is it possible to identify the surface type?

The development of the control system is intended to help provide answers to these questions, and will hopefully provide a solid robotic platform for further scientific study in the future. However, if the software system is a failure in so far as it is unable to control the ARGO, or does so with little or no control, then this project will not be considered a failure. This would simply result in negative responses to most of the research questions posed. For example, it may simply be the case that it is not possible to automate such a vehicle.

2.3 Primary Objectives

There are a number of primary software engineering objectives that this project aims to accomplish which are discussed in further detail in this section.

2.3.1 Control Software

The control software will be responsible for providing a solid and stable interface to the hardware of the ARGO. This will allow for the control of all the ARGO's low level operations, including various tasks such as the adjustment of throttle values and the positioning of the steering. It will further allow for future development of higher level control software capable of performing tasks such as navigation, obstacle avoidance etc. This section gives an overview of the features required in order to produce a software system such as the one described.

Teleoperation

The ability for a human to control the ARGO to maneuver it before and after testing or for display purposes would be highly advantageous. As such, the low level control system should have simplistic teleoperation capabilities. This should include:

- Set throttle position.
- Set handle bar position.
- Emergency stop.
- The ability to give control to a high-level control system.

Turn By Angle

The control software should have the ability to instruct the ARGO to turn by a specific angle, relative to the current heading. Such angles would be expected to be within the range of -360 to 360 degrees where the sign denotes the direction of travel, with a negative sign indicating that the ARGO should turn in an anti-clockwise direction.

Appropriate control techniques should be used to ensure as much accuracy as possible, within the constraints of the vehicle and bearing in mind the research objectives of this project relating to accuracy on differing surfaces.

Turn To Angle

The ARGO should be able to turn to any specified heading relative to North. Angles would be expected to be absolute, that is between 0 and 359 degrees. An appropriate turn direction should be chosen when performing a turn to ensure that the ARGO never spends longer than is necessary to achieve the target heading.

Once again, accuracy may be impaired depending on the constraints of the vehicle, surface type and hardware.

Move Forwards

The control system should be able to move the ARGO forwards in a particular direction, indefinitely until told to stop. In addition, the speed of the vehicle should be controllable via this interface.

Get Sensor Data

The control system should provide an interface to gather raw or formatted sensor data. Specifically this data might include:

- GPS data including longitude, latitude, altitude, number of satellites, hdop etc.
- Accelerometer and gyroscope data.
- Compass data.

2.3.2 Simulator Software

The simulator software will need to have an identical set of interfaces as the control software. However, instead of manipulating physical hardware, it will simulate the operations of the vehicle. This means that developers of higher level control systems (such as obstacle avoidance, track following etc) or scientists wishing to try out their hypothesises would be able to do so without leaving the laboratory environment.

In addition the simulation software will need to have operations to set specific target environments. This will be required in order to more accurately simulate the movements of the ARGO in differing conditions.

It should be noted that a simulator could be an entirely separate dissertation. For this reason the simulator is expected to be relatively simplistic with a limited set of features. This section details the requirements for the simulator software.

Dynamic Surface Types

The simulator should have a framework for developing different classes that each represent different surface types. Such surface type classes should hold data representing the performance of the ARGO on that specific surface. This data may include:

- Maximum and minimum speed possible.
- The speed of the vehicle at differing throttle values.
- Rates of turn and associated response times.

It should be possible to develop new surface type classes independently of the simulator using an API. As such new surface type implementations can be developed as they become needed - without needing to make modifications to the main code base of the simulator.

Load Map File

The simulator should be able to load a relatively simplistic map file. Such a map file will define differing areas, each with their own surface types. The aim is that this will allow for the development of relatively simplistic two-dimensional representations of different geographical locations.

Display ARGO

The simulator should display the ARGO on the loaded map at some defined starting co-ordinate. The ARGO should then move around the map appropriately as per the control directions received.

Simulate ARGO Command Set

All command sets from the physical ARGO should be simulated. This includes the turn by angle, turn to angle and move forward requirements.

It will be assumed that some set point in the environment will be considered North for the purposes of calculating compass and other sensor data. However, it may not be appropriate to generate some sensor data such as GPS data.

2.3.3 High-Level Example Software

In addition to the control software and simulator a very basic higher-level control system will be produced to demonstrate the functional operation of the control software. The higher-level software will have a number of operations to perform, including:

- Providing specific angles to turn through.
- Providing specific target headings.
- Providing specific target speeds or throttle settings to the vehicle.

The high-level control software will in effect operate as a 3rd party high-level control system would and will be developed for two reasons. First it will provide a excellent way to test the functionality of the main control system, and secondly it will act as an example application for future software development.

2.4 Secondary Objectives

In addition to the primary objectives there are many possibilities for enhancements and further research. This section aims to give a brief overview of *some* which could potentially be attempted if sufficient time exists.

Enhanced Simulator

The simulator software could be greatly enhanced in a number of ways:

- The use of a 3D environment to allow the simulation of more of the environment such as obstacles with differing heights.
- The use of a physics engine to more accurately simulate the movements of the ARGO.

Simple GPS Navigation

It would be nice to have the ability to tell the ARGO to navigate to a GPS coordinate. Whilst more sophisticated methods of navigation and path planning [8] may not be appropriate given the scope of this project, a simple "turn to heading and drive forward" approach may be achievable.

Obstacle Detection

Some of the research being undertaken at Aberystwyth includes obstacle detection and avoidance using various methods from camera images to 2D laser scanning. Depending on the time remaining it may be possible to incorporate existing obstacle detection algorithms and techniques to the ARGO.

2.5 Non-Functional Objectives

In addition to the functional and research objectives there are a number of non-functional requirements for the control system. These are important as they can have a significant impact on both the accuracy of any results obtained and the success of the control system.

2.5.1 Real-Time Operation

Many features of the control system will have to operate within real-time constraints. Examples of where this becomes evident include the performance of operations such as turn by angle. In such an instance it will be vital that sensor data (such as the compass) is returned to the control algorithm on time. If the reading of sensors is delayed it could result in the vehicle significantly overshooting the desired target angle due to the anticipated turn rates.

Further, if the reading of sensor data is "late", meaning insufficient readings during operations, it could potentially result in the invalid analysis of the data when identifying the effects of surface types on the characteristics of the ARGO.

In addition, a number of the existing control techniques already discussed such as PID controllers require guaranteed loop time in order to be tuned correctly. Failure to have a response time that is consistent could mean that the tuning is correct for one loop but not for another.

2.5.2 Safety

Given the size and nature of the ARGO safety is of significant importance. It is conceivable that the ARGO could seriously injury or, in extreme cases, kill someone if the software fails.

As such, the robotic platform should have mechanisms to manually override the control software in the event of unexpected or unsafe behaviour, and preferably the software should

have procedures in place to ensure that the vehicle fails safe if communications are lost with higher-level control software, or operator requests are inappropriate.

2.6 Assumptions

In order to minimise the scope of the project to more reasonable levels given the time constraints a number of assumptions have been made regarding the vehicle, software and operating environment. These assumptions are discussed here.

2.6.1 Relatively Flat Ground

The hardware being used for this project is expected to be relatively inexpensive and perhaps primitive. As such it is anticipated that the compass (in particular) will be unable to deal with significant inclines. Whilst gradual gradients and uneven ground are unlikely to cause problems, larger hills or slops could conceivably cause a number of problems:

- The slope may have an impact on the speed of the ARGO. This could result in significantly more overshoot than anticipated, or faster/slower speeds than desired.
- The slope is almost certainly likely to cause issues with the compass heading readings.

As such, it is assumed that the ARGO is on flat ground during the testing. During the course of the development solutions to these problems may be identified (such as the use of a more expensive compass) and as such this assumption may be removed as the project progresses.

2.6.2 Compass Data is Correct

Given the nature of a petrol engine there is likely to be vibrations and minor electrical interference (from spark plugs) that could cause error in the sensor data from the compass. Despite this it will be assumed that the compass data returned is correct possibly after some simple filtering.

Chapter 3

Market Analysis

"Do you realise if it weren't for Edison we'd be watching TV by candlelight?"

Al Boliska

3.1 Why the System is Needed

As already discussed in the introduction (Chapter 1) this projects primary aim is related to research currently being undertaken in New Zealand. In that respect a need had already been identified for a robotic system.

It is also clear that all robotic systems need some means of controlling them. Various approaches already exist from the traditional deliberative methods to the use of Artificial Intelligence (AI) and Artificial Neural Networks (ANN). [10, 11] Due to the massive variations in robotic systems this control software is often specialised. As a result it is clear that there is no one control system to work and fit all robotic systems. Due to the custom nature of the hardware configuration being used on the ARGO no such control system exists.

A significant portion of robotic systems have simulator environments. [5, 12] This allows for the development and testing of software, possibly without needing to use expensive hardware. In addition it allows for troubleshooting of software when the physical robot may be hundreds or, in the case of space robotics, millions of miles away. Clearly the simulator must be a close approximation of the physical hardware and control software. For this reason the simulator must either be custom built, or highly configurable such that it can allow for the building and development of custom robotic configurations. Again, due to the custom nature of the ARGO hardware and the decision to produce a custom control system, there will be a clear need for a purpose built simulator.

3.1.1 Possible Future Uses

Military

The military have been using Unmanned Air Vehicle (UAV) and Unmanned Ground Vehicle (UGV) robotic systems for many years and their use is expected to increase over the next decade. [21]

There are many possible uses for robotics in the military. Some of these are clearly not directly relevant to this project (such as UAV or airborne robotic systems). However there are many possibilities for UGVs. Some of these might include:

- Detection and disarmament of mines and explosives.
- Patrol and intrusion detection of secure areas.
- Incursion into enemy territory.
- Troop transportation, especially given the amphibious nature of the vehicle, etc.

Space Exploration

Due to the harsh environment in space, robotics are already widely used for exploration in the form or orbiters, landers and rovers. A paper [13] released in 2003 suggests many current and possible future uses for robotics in the space arena, including:

- Planetary surface exploration.
- In space assembly, inspection and maintenance.
- Surface and in-space human assistance.
- Surface instrument deployment.
- Science planning and perception.

Whilst it is *highly unlikely* that this project will directly be used for space exploration both due to the size and weight of the ARGO and due to the fact that space certified components are not likely to be used in its construction, some aspects of this project such as the simulation environment and the control software may prove useful as a base for future research in the field.

Research & Teaching

The Intelligent Robotics Group (IRG) at Aberystwyth University, one of the largest robotics groups in the UK, are constantly pursuing new ways of using and controlling robotic systems. As such in addition to the research being conducted in New Zealand it is highly likely that the ARGO will be used for additional research in various areas after the initial research is completed. This is further likely due to the fact the department owns two ARGOs.

According to the IRG website[1] their research topics include biologically inspired robotics and control, space robotics, visual navigation and mapping and field robotics. Some of their research has included development of the European Space Agency (ESA) ExoMarse rovers, calibration of the robotic arm on Beagle 2 and the development of Beagle-B, a robotic sailing boat.

In addition to their often internationally leading research, many of the robotic systems at Aberystwyth are used by students at the university for teaching, dissertation and PhD research. Having a robotic platform available which can easily be controlled and adapted to include additional sensors according to a specific goal would be highly valuable to the department.

[1]IRG website: `http://www.aber.ac.uk/en/cs/research/ir/`, Last Accessed: 16th October 2009

3.2 Existing Products

3.2.1 Control Software

Player

Player[2] is an open-source network control interface for robotic systems. In effect, Player sits between a client application, which is generally used to do high-level robotic control such as path planning, and a platform specific hardware driver used to physically control the motors and sensors on the robot. Player then provides a network interface for the control of the robot, allowing the high-level control software to be developed in any programming or scripting language that supports network communication.

Due to the modular nature of the software it can be used to control almost any robotic system simply by the production of a driver. As such Player has the advantage that it can allow for one control system developed for one robotic system to be used on any robot that has a Player driver. This also means that Player can be used with various simulation environments (detailed later in this chapter) by simply producing an appropriate driver for conversion of Player commands to the simulator commands.

Player is already widely used within the IRG at Aberystwyth and this means that is could potentially be a competitor to aspects of this project. However, it is also possible that a Player driver could be developed which would allow for Player to control the ARGO using the control software developed for this dissertation. Whilst this would lead to another layer of abstraction from the physical hardware and the high-level control software, it would also mean that some of the existing control software and algorithms could potentially be used on the ARGO with few or no modifications.

Bespoke Control Software

Due to the nature of the robotics industry there are many bespoke software solutions to control robotic systems. Some of these use traditional methods often known as deliberative control systems, others use a variety or hybrids of different methods such as reactive control systems, ANNs and Artificial Endocrine System (AES).

There can be many reasons for the development of such control software, some of which may include:

- Research into new or better methods of controlling robotic systems.
- Research into other areas such as biological systems. This can then lead to the simulation of these biological systems through robotics.
- New or bespoke hardware that doesn't already have control software, such as with this project.

The problem with such software is that it tends to be developed for a specific robotic system, or it is designed to perform a specific task. As such any existing software of this kind will be unlikely to function properly or adequately for the ARGO.

[2]Player website: `http://playerstage.sourceforge.net/index.pgp?src=player`, Last accessed: 18th October 2009.

3.2.2 Simulator Software

Various simulators for robotic systems exist. This section aims to give a brief overview of some of these simulators and their features.

Stage

Stage[3] is a relatively simple open-source 2D robotic simulator (Figure 3.1) designed for use in a UNIX/Linux environment. According to the Stage simulator website the aims of the project are to "support research into multi-agent autonomous systems." As such the resulting software attempts to simulate a wide range of robotic systems, often simultaneously, to see how they react together. However, the models used for the robotic systems tend to be relatively simplistic and "computationally cheap."

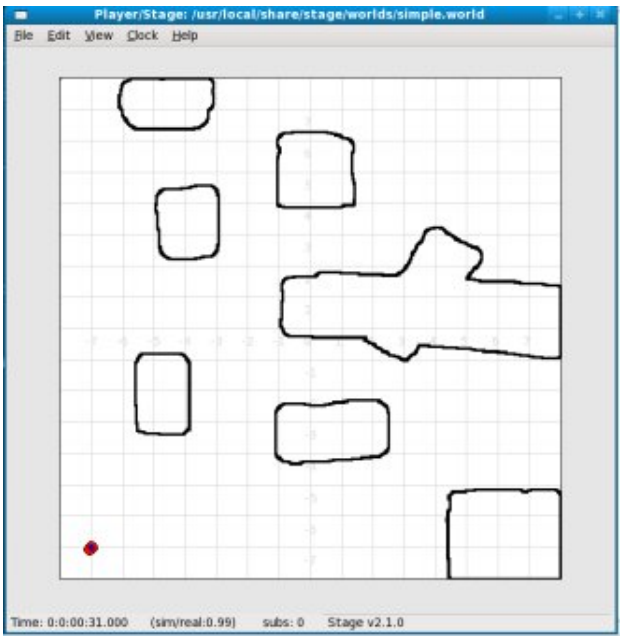


Figure 3.1: The Stage Open-Source 2D Robotic Simulator Environment

Stage can work using one of two methods:

1. As a Player plug-in allowing code developed for robots that use Player to seamlessly use the simulator environment. The advantage of this approach is that most of the time code developed using the simulator can be used on physical hardware with few or no modifications.

2. Via the provided C library (libstage). Unfortunately using this method means that once code is produced for the simulator it will need modifying to work with the libraries provided by the control software of the physical robot, introducing the possibility of mistakes during the conversion from simulator to the physical world.

[3]Stage website: `http://playerstage.sourceforge.net/index.php?src=stage`, Last accessed: 18th October 2009.

Gazebo

Gazebo[4] is a more sophisticated 3D robotic simulator environment (Figure 3.2) which is still under heavy development by the same group that produce Player and Stage. Gazebo is also an open source project designed for use on a Linux/UNIX system. Unlike Stage, Gazebo does not intend to simulate a large population of robots, but a smaller population more accurately.

In addition to simulating the robotic systems, Gazebo can also simulate a lot of the hardware that robots use including sonar, GPS, stereo cameras etc. The environment also includes a physics engine which allows for much more complex simulations such as picking up objects or pushing objects around the simulated environment.

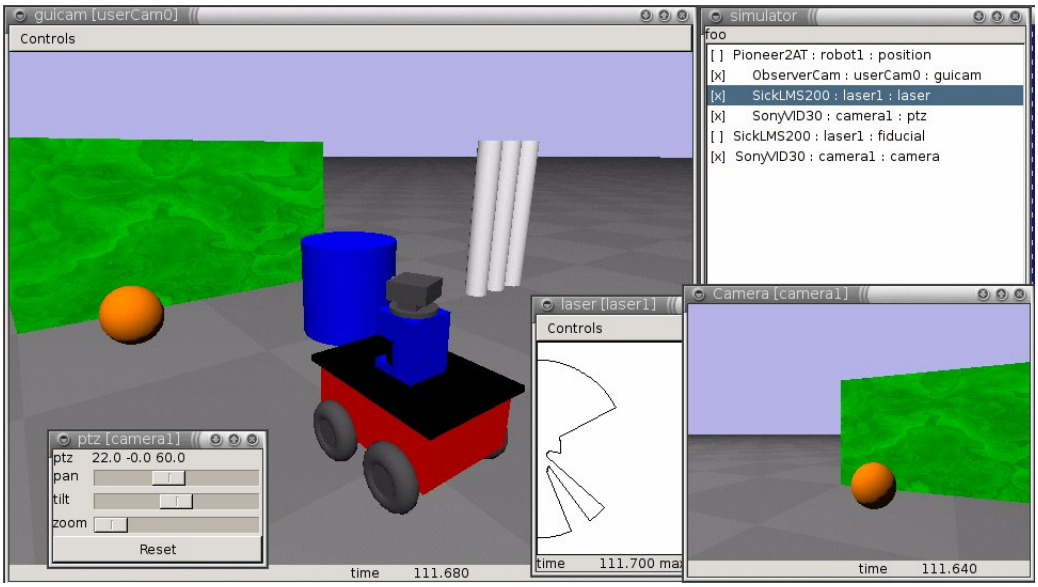


Figure 3.2: The Gazebo Open-Source 3D Robotic Simulator Environment

An additional advantage of Gazebo is that, as well as providing various templates for existing robotic systems such as the Pioneer[5] robots, it also allows developers to build custom robot configurations and then simulate their movements and operations within the environment.

As with Stage, software developers can interact with Gazebo using two methods. The first being a Player plug-in, and the second being a C library called libgazebo. As such, both these methods suffer from the same advantages and disadvantages already identified with the method of interaction with Stage.

Simbad Java 3D Robot Simulator

Simbad[6] is a 3D robotic simulator written in Java (Figure 3.3). According to the Simbad website the simulator has been designed specifically for educational and scientific purposes.

[4]Gazebo website: `http://playerstage.sourceforge.org/gazebo/gazebo.html`, Last accessed 18th October 2009.

[5]Pioneer robot: `http://www.activrobots.com/ROBOTS/p2dx.html`, Last accessed: 18th October 2009.

[6]Simbad webiste: `http://simbad.sourceforge.net/`, Last accessed: 18th October 2009.

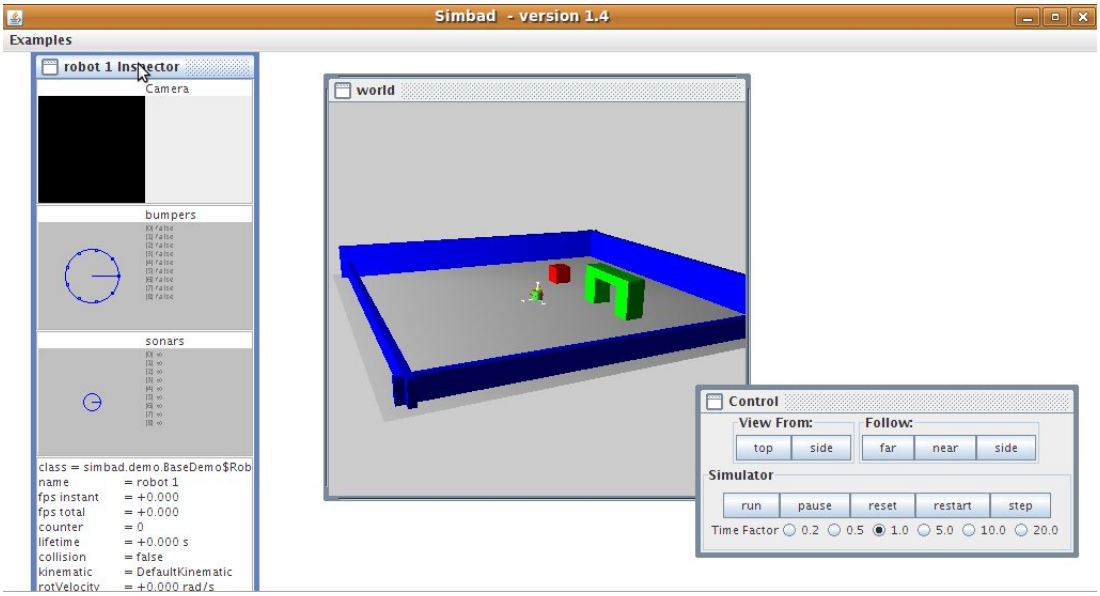


Figure 3.3: The Simbad Java 3D Robotic Simulator

Simbad boasts various features including the ability to easily develop control software, modify the environment and the use of various sensors including color monoscopic cameras, sonars, Infra-red (IR) and contact sensors such as bumpers.

As with the other simulator environments, Simbad is an open source product. However, unlike the other simulators discussed, Simbad has the advantage that it has been developed in the Java programming language. This means that it can be used on any operating system that supports the Java Runtime Environment (JRE).

Chapter 4

Risk Analysis

"Foolproof systems don't take into account the ingenuity of fools."

Gene Brown

4.1 Overview

There are a number of different risks that could have an impact on this project. Such risks could happen during the development of the project and could therefore have an impact on the success of this project. Other risks are identified which relate to the operation of the ARGO, and as such clearly affect how the software should operate.

This chapter aims to give an overview of these different risks along with suggestions for their reduction or total mitigation.

4.2 Development Risks

4.2.1 Hardware Development Time

Identified Risk

One major aspect of this project requires the ARGO to be equipped with various hardware components. At the time of starting the project these components are on order. It is expected that the time for delivery will be approximately three weeks. After this point it could take another two to three weeks to install the hardware. The result of this is a potential delay of six weeks.

Risk Mitigation

A number of things can be done to reduce this significant risk:

- Aspects of the simulation software can be started, and possibly completed, before the hardware is delivered. Although this will not reduce the risks related to the hardware, it will mean that time otherwise 'wasted' can be put to effective use. This would mean that in the later stages of the project the coding can focus primarily on the physical aspects rather than the virtual.

- Prototype hardware already available such as accelerometers, GPS etc can be used with the ARGO to get initial readings for the development of the simulator. This could also potentially help towards a more appropriate choice of hardware. For example, it may be found that the accelerometer or compass components are not accurate enough during testing with the prototype hardware. With this information it will be easier to specify more appropriate components.

4.2.2 Hardware Failure

Identified Risk

All hardware is subject to failure. Should the hardware fail this could have devastating consequences for the success of this project.

Risk Mitigation

- In order to allow for possible hardware failure it is hoped to have major aspects of the programming completed by December 2009. This way should hardware fail there will be sufficient time to order replacement components before the final project is due.
- The simulator developed could potentially be used to demonstrate some of the high-level aspects of the project (the high-level example control software for example) should the ARGO be irreparable.
- Many components to be used in the production of the ARGO will be ordered in duplicate, or there are already spares available. This will mean that most hardware failures should be relatively simple and quick straight swap replacements.

4.2.3 Access to Hardware

Identified Risk

The hardware being used is located on a university campus approximately 20 minutes walk from the main campus. It is not always staffed and as such there will almost certainly be periods of time where access to the physical hardware is limited.

Risk Mitigation

- It is intend that the time when hardware is not available can be used to complete work on the simulator aspect of the project. Further, documentation could also be maintained during these periods.
- Where possible the use of the simulator environment, once completed, can be used to conduct tests and further development work on the high-level example control software until the hardware is once again available.

4.2.4 Lack of Previous Experience in the Field

Identified Risk

The author of this report has limited experience of the robotics field and as such some of the terminology and methods may be less familiar. This could potentially lead to unwise design

and development decisions both with respect to the hardware and sensor requirements and to the software development.

Risk Mitigation

- Investigative work will be conducted to more thoroughly understand any aspects of the project where the author lacks experience.
- Existing hardware platforms can be used as prototype hardware to gain readings and measurements of the ARGOs performance in order to assist in the decision making process as to what sensors and hardware will be required.

4.3 Operational Risks

4.3.1 Software Failure

Identified Risk

The software controlling the ARGO may fail or attempt to perform operations which are not sensible or worse, dangerous.

Risk Mitigation

- The control system software should have sufficient protective methods to prevent such occurrences in the first place. This should include the use of appropriate coding standards, recognised design and development methodologies, appropriate error handling and validation of parameters to functions and fail-safe mechanisms.
- All testing of the hardware will be conducted with more than one person; preferably one person will remain in the vehicle at all times allowing for the emergency manual application of the brakes or the turning off of the engine.
- The use of a industrial grade electronic remote override will be installed to cut power to the engine remotely if all other safety systems fail.

4.3.2 Hardware Failure

Identified Risk

During the course of normal operations hardware components may fail. In such circumstances software control of the ARGO may become impossible. For example, failure of a servo controlling the throttle would not be rectifiable by software. Attempts to reset the throttle to an appropriate level would fail due to the damaged hardware.

Risk Mitigation

- As with the software failure, the industrial grade emergency shut-off will be helpful in such circumstances.
- Once again, as with the software failure, more than one person should be present during all tests to provide assistance or manual override of breaking etc.

Chapter 5

Development Methodology

"A computer lets you make more mistakes faster than any invention in human history - with the possible exception of handguns and tequila."

Mitch Ratcliffe

5.1 Choosing a Methodology

When working on any engineering development it is prudent to have a recognised, accepted and proven methodology to control and manage the development process. This is as equally important in the software engineering world as it is for any other engineering discipline. Having a good solid methodology provides numerous benefits during the software development process, including the control and monitoring of the project progress and ensuring that all team members have a consensus as to the direction of the project.

There are a number of differences between the software engineering community and that of other engineering discipline such as mechanical engineering. To use a simple example of building a bridge. In mechanical engineering it is very important to have a detailed design of the bridge before the manufacturing of parts and assembly work can commence. This means everything must be specified down to the smallest detail. One screw with the wrong thread or the wrong diameter could potentially cause the whole bridge to collapse. As such, engineering methodologies have emerged which enable engineers to specify, and build their bridges such that mistakes like the size or type of a screw or bolt do not happen. Adaptations of these methodologies are then often used in the software engineering industry leading to methodologies such as the waterfall methodology. [15]

However, unlike with mechanical engineering where it is clear that the bridge must span a certain area, and be load-bearing up to some maximum figure etc, software is not a tangible product and is subject to change. In software engineering it is not always possible to know all of a users requirements up front or in great detail. This can mean that projects developed using traditional methods can often go over budget, take longer than expected or even fail all together. Others can find that projects using such methodologies deliver working software on time and in budget. For this reason, over time, various software engineers have developed new and differing methodologies for the development of software. Boehm suggests in his paper [3] that these methodologies follow a spectrum (Figure 5.1) from the traditional heavy-weight plan-driven methodologies such as the waterfall or spiral methodology to what is known in the industry as agile methodologies such as SCRUM, [17] Feature Driven Development (FDD) [1] and eXtreme Programming (XP). [2]

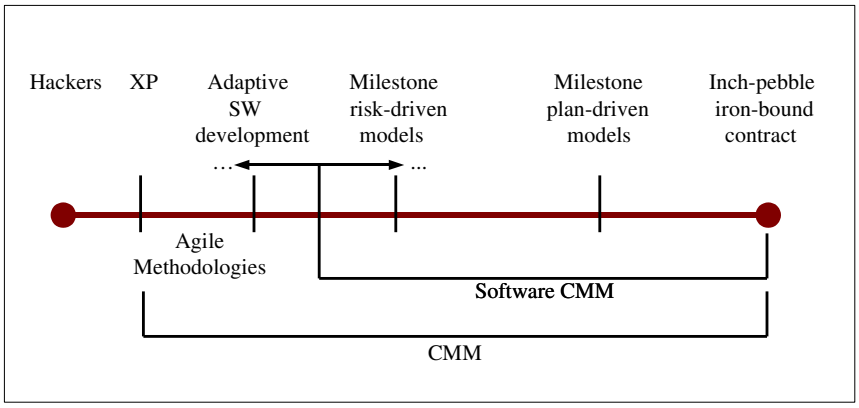


Figure 5.1: Boehm's Planning Spectrum

Choosing which methodology is appropriate for a given project is a difficult task that is aided by the experience of the engineers undertaking the project. For example, an engineer may have experience of using various different methodologies, and have their own ideas about which methodologies work under certain circumstances. Further, the choice of methodology also depends on the amount of up-front information known about the end product. If engineers have worked on similar projects previously, and have a good idea of the requirements, methodologies such as the waterfall methodology may be appropriate as more accurate estimates and identification of tasks and milestones can take place early on in the development stage. However, if little information is known, or the project is subject to change, or there is a lack of experience and thus significant risk involved then agile methodologies can often prove to be useful as they allow for a changing customer requirements during the project. If nothing else, one thing is clear - no single methodology exists that will suit all development projects.

In order to decide on a methodology for use during the project it was necessary to identify various aspects of the project. These aspects are discussed briefly.

5.1.1 Known Requirements

It is clear up to this point that there are various requirements which are known and almost certainly will be set in stone. These are detailed in the project objectives (Chapter 2). This may suggest that a up-front design methodology such as the waterfall methodology would be appropriate for this project.

It is also clear that there are a lot of unknown aspects to the project, and further that some of the requirements which do exist could be considered vague. With this in mind it is highly likely that the requirements could need modifying during the course of the project, and this suggests that an agile approach would be more appropriate for this project.

5.1.2 Authors Experience

One risk identified during the risk analysis (Chapter 4) was that the author has limited experience in the field of robotics. This means that it is highly unlikely that the author has

sufficient knowledge to accurately take into account all possible influences on the project and as such plan mile-stones for the project upfront as is required by a heavy-weight methodology.

It is also likely that a significant amount of spike or research work will need to be conducted in order to bring the author up to speed. Given this, it suggests that an agile methodology would be most appropriate.

It is further the case that the author has had experience of some agile methodologies such as XP and also of plan-based methodologies such as the waterfall methodology. Whilst both projects were a success, delivering the software products on time and working, it was felt that the agile approach worked better for the author and was more suited to the authors personality. This further enhances the suggestion that an agile approach would be appropriate.

5.1.3 Project Size

Given the nature of this project, it will be conducted by a single person - namely the author. Plan driven methodologies are often more suited to large teams of people where communication of information is only viable via documentation, wikis, forums and other project management tools.

Agile methodologies are much more suited to smaller teams. This once again suggests that the author should take an agile approach to the development of the software.

5.2 SCRUM

After reviewing the factors affecting this project it has been decided that an agile approach would be most suited. Numerous so called agile methodologies exist, and of these SCRUM has been chosen as the methodology for developing this project.

SCRUM is a iterative framework consisting of a number of different practices and roles for the development and management of software products. SCRUM developed over a number of years separately in various organisations. [19, 6, 18] SCRUM provides a nice balance between plan-driven and extreme agile methodologies, allowing for some up-front design and requirements analysis whilst still managing to be agile in so far as it is able to account for and adapt to change during the project.

5.2.1 SCRUM Roles

There are a number of roles defined in SCRUM which can be categorised into one of two groups: pigs or chickens. The 'pig' roles are those who are fully committed to the project such as the developers. The 'chicken' roles are those who are not actually part of the process but those who must be taken into account such as managers and customers. In this section a brief overview of the primary roles in SCRUM is given.

SCRUM Master

This is a member of the team who maintains the process. That is, he or she is in-charge of ensuring that all team members are following the SCRUM process. The Scrum Master also has various other duties during the different aspects of the SCRUM process such as controlling the stand-up meetings discussed later in this section.

Traditionally the SCRUM Master role is taken on by a member of the team and is therefore considered to be a 'pig'. However, it is important to note that the SCRUM Master is not the leader of the team as, in SCRUM, the team is supposed to self-organise.

Given that this is a one-man project this role effectively becomes superfluous to requirements and so will be ignored during the course of this project.

Product Owner

This is a person who represents the customers and stakeholders. The product owner is there to ensure that the development team work on the tasks that are most important to the customer. Whilst the customers are considered to be 'chickens' in SCRUM, the product owner is considered to be a 'pig'.

In this project, there are various candidates for the product owner role:

- The project author is clearly a stakeholder as he has to produce the software in order to gain a degree.
- The project supervisor is a customer as he wishes to use the end product at a later date for future research.
- The research team in New Zealand are also customers as they eventually hope to use the end product to control the ARGO in their possession.

During the development process it is the product owner who has the last say as to which items in the feature or product backlog should be developed next. For this project it is intended that the author will act as the primary product owner.

The Team

The final primary role in the methodology is 'the team'. This includes the engineers who do the actual design, analysis, implementation, testing, etc during the development process. All members of the team are considered to be in the 'pig' category.

Once again, given the nature of this project the 'team' will consist solely of the author.

Stakeholders or Customers

The stakeholders or customers are the people that the software is being developed for. They are considered to be 'chickens' in the SCRUM methodology.

The Product Owner is responsible for contacting the stakeholders and identifying their needs. As such, anyone who could potentially be a product owner in this project could also be a stakeholder. Specifically this means this project has two stakeholders:

- The author.
- The project supervisor.

Although the team in New Zealand would eventually like to use the resulting robotic platform, they are not considered to be a primary stakeholder or customer for this project. The only contact the author has to the team in New Zealand is via the project supervisor, and so for the purpose of this project the project supervisor speaks for all the stakeholders.

Managers

The managers are people who setup and maintain the development environment etc. In SCRUM the managers are considered to be 'chickens'. In this project the author is clearly the primary person who decides which development environment will be used.

5.2.2 SCRUM Phases

SCRUM is split into three separate phases (Figure 5.2), each of which is detailed in this section.

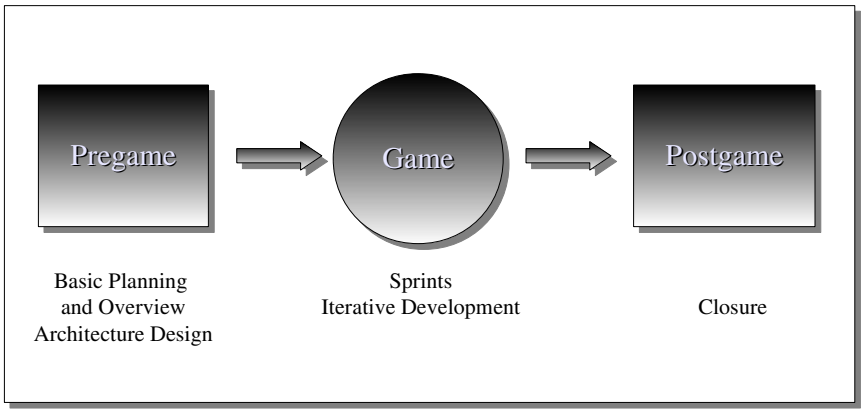


Figure 5.2: SCRUM Development Methodology Phases

Phase 1: Pregame

The pregame phase is used to do a some initial planning and design work. The pregame stage can also be used to do spike or initial investigative work to help identify and minimise risk during the project.

Whilst SCRUM is an iterative methodology it is still important to have a certain amount of planning involved. In SCRUM this involves a number of tasks:

- Developing a backlog list. The backlog is effectively a list of features that are known to be needed.
- Defining the delivery date and functionality of one or more releases.
- Selecting the release most appropriate for immediate development.
- Defining project teams.
- Assessing risk (through spike work).
- Development tool and environment selection.
- Various management tasks such as marketing analysis, estimating cost, marketing, and ensuring that funding is available.

It is clear from this list of tasks that some are inappropriate for this project such as ensuring that funding is available and defining project teams. As such, these stages will not be used during this project.

Once the planning is completed an architecture or high-level design is produced. It is important to note that because the methodology is iterative the design work completed should not be heavy-weight as is the case with plan-based methodologies. That is to say, the design should be concise and show a general overview only of features that are known and the expected work involved in producing those features. Further, it is acceptable (and expected) that the design can change considerably during the development of the project.

The design and overall system architecture produced during the pregame stage can be considered a starting point. It will help with the identification of tasks and the identification of how many sprints are needed etc. During each sprint in the game stage the design is extended, modified and improved.

Phase 2: Game

The game stage is the main aspect of SCRUM. It is during stage that the design and development work really takes place in the forms of sprints.

At the beginning of each sprint a 'sprint planning meeting' takes place where a number of things are discussed including the work that is to be completed, preparing the sprint backlog which details the time it should take to do the work. Given the nature of this project, this meeting will take a written form at the start of each sprint chapter.

Once the initial meetings are completed the sprint begins. Each day during the sprint a 15 minute meeting is held. During this meeting each member of the team discusses what they have achieved since the previous day, what they intend to do that day and any problems that are preventing them from accomplishing their goals. Once again, due to the nature of the project, this meeting will not take place. Instead the general progress of the sprint will be documented in the appropriate section of this report.

In larger projects using SCRUM it may be the case that work is distributed amongst many teams. After the Daily SCRUM meeting a 'SCRUM of SCRUMS' meeting takes place amongst team leaders with a similar agenda to that of the daily SCRUM, but discussing the progress of teams as opposed to individuals. As this project does not have teams of people this meeting will be omitted.

At the end of each sprint a Sprint Review meeting takes place where a review takes place of the work that was completed, and not completed. Anything that can be demonstrated to the stakeholders is. After this meeting a sprint retrospective takes place where all team members reflect on the sprint and try to continually improve the development process by answering two questions: What went well during the sprint? What could be improved in the next sprint? These items will appear at the end of each sprint chapter.

Phase 3: Postgame

The final stage of the project is postgame. This happens when there is no more time available for development and those features in the backlog of most importance as decided by the project owner have been completed.

At this point a number of final tasks are completed, designed to prepare the product for general release. This means the integration, testing, user documentation, training material and marketing material are produced.

Chapter 6

Phase 1: Pregame

"I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones."

Linus Trovalds, The launch of Linux.

6.1 Environment Setup

One important task involved in fulfilling phase one is to ensure that a functional development environment exists. This includes many aspects such as ensuring backups are taken, that source code is under version control systems, that all compilers and libraries are at the correct versions etc.

This section details a number of the decisions taken in order to achieve a suitable working environment for development.

6.1.1 Version Control System

Version control is very important in the management of a software project. It allows tracking of all changes made to software. This means that if mistakes are made which cause the software to fail, it is easy to retrieve a previous working version and identify the changes that caused the faults.

The author has had experience of various version control systems including Subversion[1], Mercurial[2] and GIT[3].

Using this previous knowledge it was decided that Subversion would be the most appropriate choice for this project. Whilst it does not offer some of the collaborative and distributed features that GIT and Mercurial offer, these are not necessary for this project and so would have simply added more complications.

Various Subversion servers were available to the author for use, including:

- Departmental Subversion Server
- Google Code
- Personal Virtual Private Server

[1]Subversion website: `http://subversion.tigris.org/`, Last accessed: 20th November 2009.
[2]Mercurial website: `http://mercurial.selenic.com/`, Last accessed: 20th November 2009.
[3]GIT website: `http://git-scm.com/`, Last accessed: 20th November 2009.

- Personal Laptop/Desktop Computer

Each of these have their advantages and disadvantages when being used. The departmental server is good choice as it is maintained by the department, backed up regularly and if there are any problems with it the department will be aware of them immediately.

The Google Code option was also of interest as it provides a management interface that can be used for issue and ticket tracking. However, Google Code requires that all projects be released under an Open Source license. Whilst the author is not opposed to Open Source (and indeed favours it), it was unclear what the position was of the university with regards to ownership of the resulting project code. As such, Google Code was not chosen.

The advantage of using both the personal virtual private server or a personal laptop/desktop computer is that the author would have full control over all aspects of the configuration and management of the subversion repository. Unfortunately it would also mean that any hardware or data corruption would be entirely the authors fault, and as such this would not provide adequate protection during the course of this project.

Given these factors a decision was made to use the departmental subversion server.

6.1.2 Backup Management

Backups are a vital part of everyday computing. The ease with which programs and documents can be changed, deleted and created means that if backups are not kept it is highly likely that at some important documents will be lost or possibly corrupted.

Fortunately, due to the authors choice of subversion server there are already regular backups being taken by the university. However, for the sake of paranoia the author has elected to produce a cron script which checks out a copy of the subversion repository each day to a separate off-site server. A such at most the author will loose one days work should all the servers in the university fail and the backup procedures also fail.

6.1.3 Language Choice

The nature of this project means that there are effectively two target environments that need to be considered. The first is the physical hardware which runs on the ARGO. The nature of the hardware to be used limits the language choice to assembly and C. Fortunately the author has had extensive experience of both languages and as such it has been decided that C will be used where possible, but if Central Processing Unit (CPU) or Peripheral Interface Controller (PIC) registers need to be modified, assembly can also be used.

The second target architecture relates to the simulator. It would be preferred that the simulator will work on various different environments and as such it is suggested that the Java programming language be used as this allows for extensive portability across any platform that provides a JRE.

6.1.4 Operating System and IDE Choice

The author has extensive experience of the UNIX and Linux operating systems and has decided to use these for the development environment. Specifically the author will be using four different operating systems during the development process.

- Gentoo[4] Linux is installed on the authors home workstation.

[4]Gentoo website: `http://www.gentoo.org`, Last accessed: 21st November 2009.

- Ubuntu[5] Linux is installed on the authors personal laptop.
- Fedora[6] Linux is installed on the workstations in the digital systems laboratory and also on the computer system in the robotics laboratory on the Llanbardan campus.
- OpenSolaris[7] (UNIX) is installed on various workstations in the department.

All these operating systems provide a similar set of features and all are familiar to the author. The advantage of having a diverse set of operating systems for the development work is that the simulator software in particular can be tested on multiple platforms to ensure compatibility.

The author is an avid user of vi/vim[8], and as such this tool will be used for the development of code. vi/vim also has the advantage that it is found on almost all UNIX/Linux systems.

For compilation the author will be using the GNU Compiler Collection[9] for compiling the C code and the Sun Microsystems Java Development Kit (JDK) Version 6.0[10] for compiling Java code.

Specifically GNU Make[11] makefiles will be used to automate the build process of the C code and Apache ANT[12] build files will be used to automate the build process of the Java code.

6.2 Investigative Work

During phase one of this project a large amount of investigative or spike work has been conducted in order to alleviate a number of risks identified in the risk analysis (Chapter 4) and in order to better estimate and plan the subsequent sprints. This work is detailed in this section.

6.2.1 Hardware Development

One of the first things that was done was to start developing prototype hardware. There were two reasons for this:

1. It was recognised early on in the project that in order for the development of the simulator code, information would be needed regarding the timings and accuracy of the ARGO during operations such as turning through differing angles etc. In order to gain this information sensor data was required.
2. The hardware was identified as a major risk and potential show-stopper for the project. As such, any work that could be done early on in the project to reduce this risk was considered to be beneficial.

[5]Ubuntu webiste: `http://www.ubuntu.com`, Last accessed: 21st November 2009.
[6]Fedora website: `http://www.fedoraproject.com`, Last accessed: 21st November 2009.
[7]OpenSolaris webiste: `http://www.opensolaris.com`, Last accessed: 21st November 2009.
[8]vi/vim website: `http://www.vim.org`, Last accessed: 22nd November 2009.
[9]GNU Compiler Collection website: `http://gcc.gnu.org/`, Last accessed: 22nd November 2009.
[10]Sun Microsystems JDK website: `http://java.sun.com/javase/downloads/index.jsp`, Last accessed: 22nd November 2009.
[11]GNU Make website: `http://www.gnu.org/software/make`, Last accessed: 22nd November 2009.
[12]Apache ANT website: `http://ant.apache.org`, Last accessed: 22nd November 2009.

Fortunately the author was aware of some existing hardware (Figure 6.1) solutions under development by the department for a number of second and third year undergraduate modules. The boards offered various sensors, including:

- 3-axis Accelerometer
- GPS Module
- Compass
- PICDem Board with a 10-bit Analog to Digital Converter (ADC)
- GUMSTIX board with WiFi and RJ45 network connections

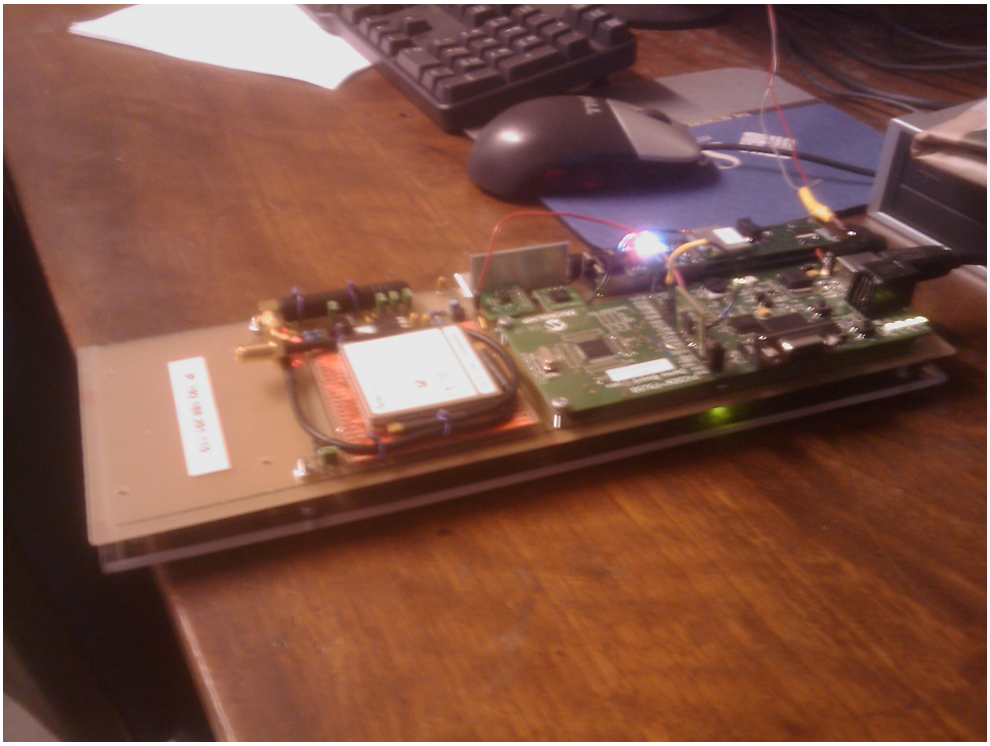


Figure 6.1: Initial Hardware Development with 2-Axis Gyroscope

It was initially expected that this configuration of hardware would provide sufficient sensor data to accurately calculate the required information for the later development of the simulator and control software. This proved (after testing the ARGO, detailed later in this chapter) to be incorrect and so a 2-axis gyroscope was also added to the hardware in such a way that it could easily be removed for the undergraduate modules.

With the hardware completed and ready for testing, spike code was produced to get various sensor readings, including the GPS location and compass heading of the ARGO. In addition, code was also written to get information from the gyroscope and accelerometer.

6.2.2 ARGO Testing & Results Analysis

With the prototype hardware complete the next step was to manually drive the ARGO, performing various different turns and movements, whilst retrieving data. Given the nature of the hardware, luggage straps were used to attach the hardware to the underside of the seat of the ARGO (Figure 6.2).

Figure 6.2: Picture of the ARGO During Testing

In total four attempts were made to get the required data. Each of these tests are detailed here.

Test One: Friday 23rd October 2009

During the first test a number of aspects were considered including the moves that needed to be made so that the author could get sufficient information for the development of the simulator and possible further testing.

It was decided that a set of 90 degree moves would be executed on differing conditions (specifically grass, gravel and tarmac) followed by a number of 45 degree turns (Figure 6.3).

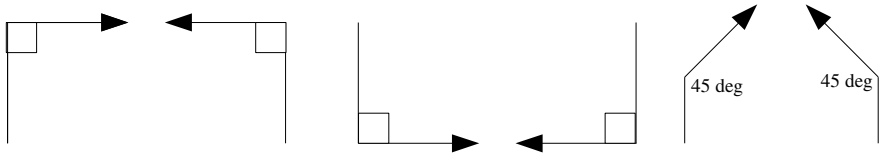


Figure 6.3: The moves executed during the testing of the ARGO

Between each test a large gap of approximately 30 - 60 seconds was left in order to identify the start and end of each test in the resulting data file. The data was then plotted

using GNU Plot[13], and the results of that are shown (Figure 6.4).

A number of things are immediately obvious from the result data. The first is right at the end at approximately 21500 readings. This sudden drop is where the seat that the test hardware was strapped to was lifted at the end of the testing and so can be ignored.

Additionally, it is also evident that the data is very noisy and does not clearly show the different turns executed. Further, because no time information was taken between readings it was not possible to accurately work out where one test ended and the next began.

Armed with this information it was clear that a gyroscope would be needed as well as an accelerometer, and so the hardware was modified as detailed earlier.

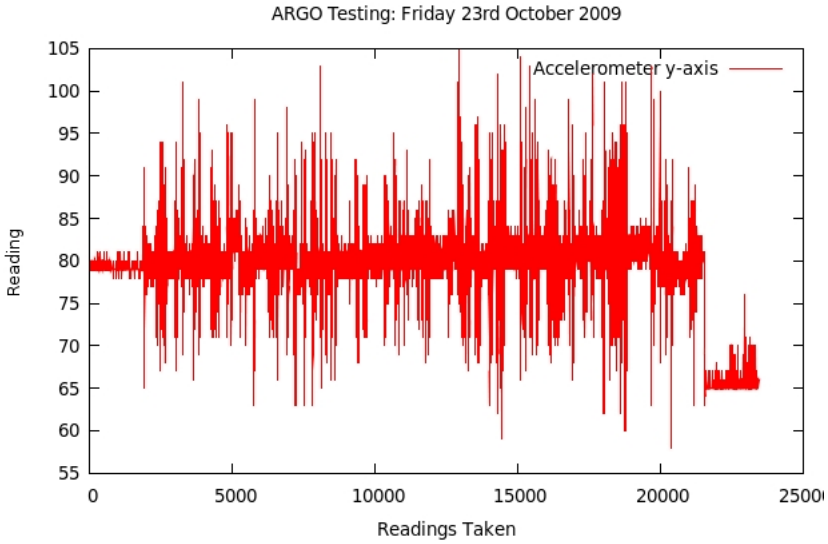


Figure 6.4: Test Result Data: 23rd October 2009

Test Two: Tuesday 10th November 2009

For the second test a similar set of movements were executed with a few additions. First, a 180 degree turn left followed by a 180 degree turn right, and then a 360 degree turn left and right. Additional information was also gathered regarding the maximum speed of the vehicle which turned out to be approximately 15 - 20 mph.

In addition to the extra data recorded, repeat readings were taken for every maneuver in order to verify the results were correct and once again the results were plotted using GNU Plot (Figure 6.5).

Further to recording the turns taken, this time the time was noted at the start and end of each test and as such it is now clear that the sampling rate was 5 readings per second. With the gyroscope data and the extra timing information the author was able to identify specific maneuvers in the data.

Figure 6.6 clearly shows four turns. The first two turns are 180 degrees, first left then right. The second two turns are 360 degrees - again first left the right.

It is easy to identify on the graph how long each turn took and to see the different readings from both the gyroscope and accelerometer during these turns and maneuvers.

[13]GNU Plot website: `http://www.gnuplot.org`, Last accessed: 20th November 2009

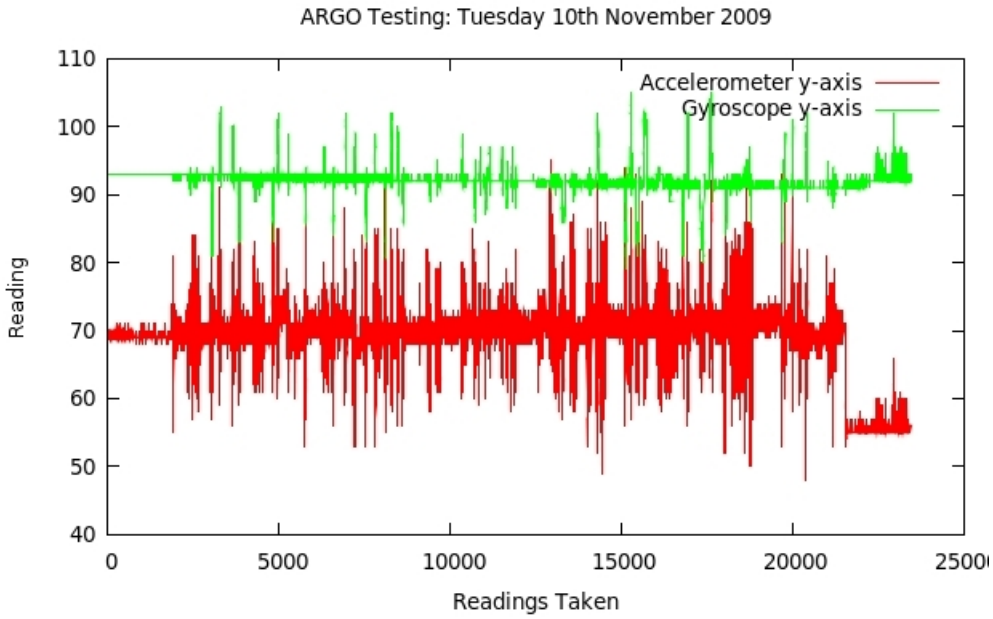


Figure 6.5: Test Result Data: 10th November 2009

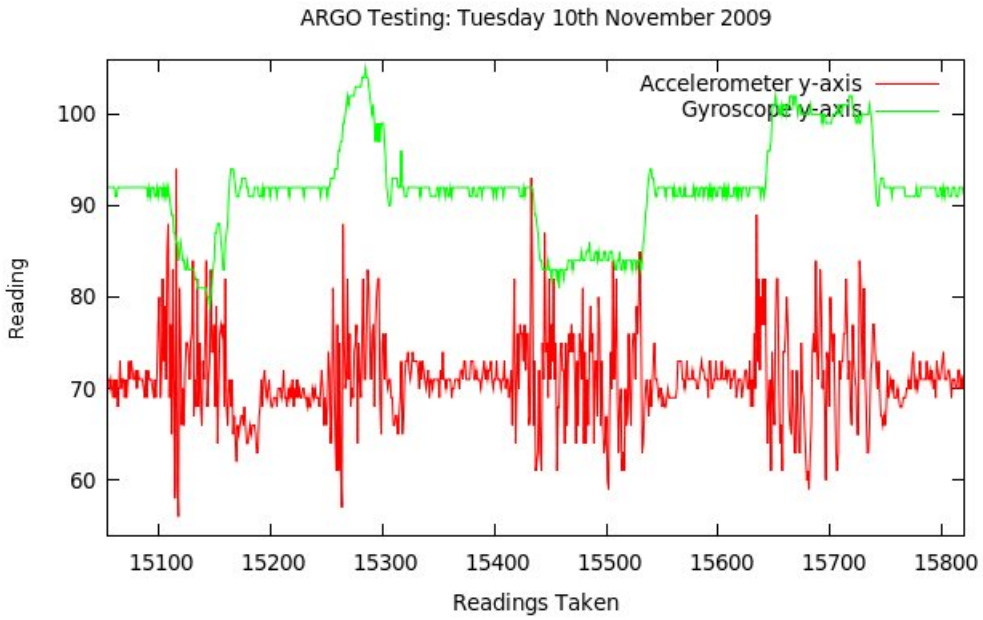


Figure 6.6: Detailed Test Result Data: 10th November 2009

Although the data retrieved proved to be useful it is also lacking as it does not show the compass data. The compass data is important as it helps to accurately identify the actual turn angle. Given that the ARGO was being manually driven it may be that a 90 degree turn was actually 85 or 100 degrees. With the compass data it would have been possible to accurately measure the turn angle.

Unfortunately during the first test the time taken to read the compass was approximately 1.05 seconds per reading. This is clearly not a fast enough sampling rate to get the required data. As such, the author spent some time attempting to identify the problem. After some investigation the problem was identified (a sleep() statement that had been put in during testing) and was fixed ready for the next day of testing.

Test Three: Wednesday 11th November 2009

With all the previous faults and errors identified and fixed this test was used to gather much more information including GPS, compass, gyroscope and accelerometer data.

Once again the same set of maneuvers that were used for the second test were used for this test. Unfortunately due to corruption in the data (Figure 6.7), which was due to the quality of the spike code, the data proved to be useless for this project and so will be ignored completely.

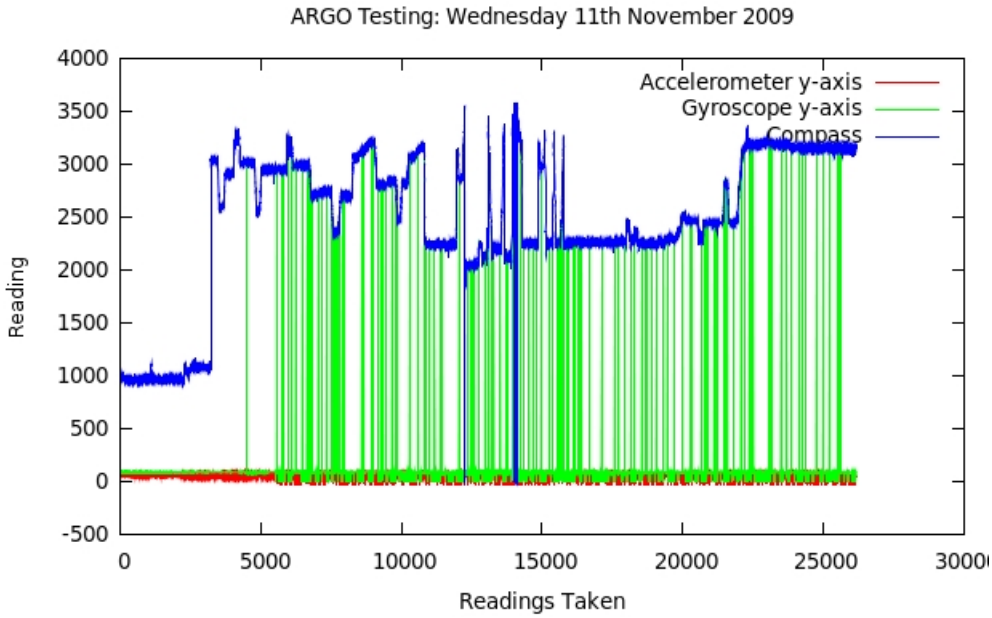


Figure 6.7: Corrupted Test Result Data: 11th November 2009

Test Four: Thursday 26th November 2009

In the fourth and final test all the aforementioned turns were executed twice so as to get repeat readings for all sensor data and the results are shown (Figure 6.8).

It is important to note a number of things regarding the data shown. First, the compass readings have been scaled by a factor of 10 so that they are within the correct value range

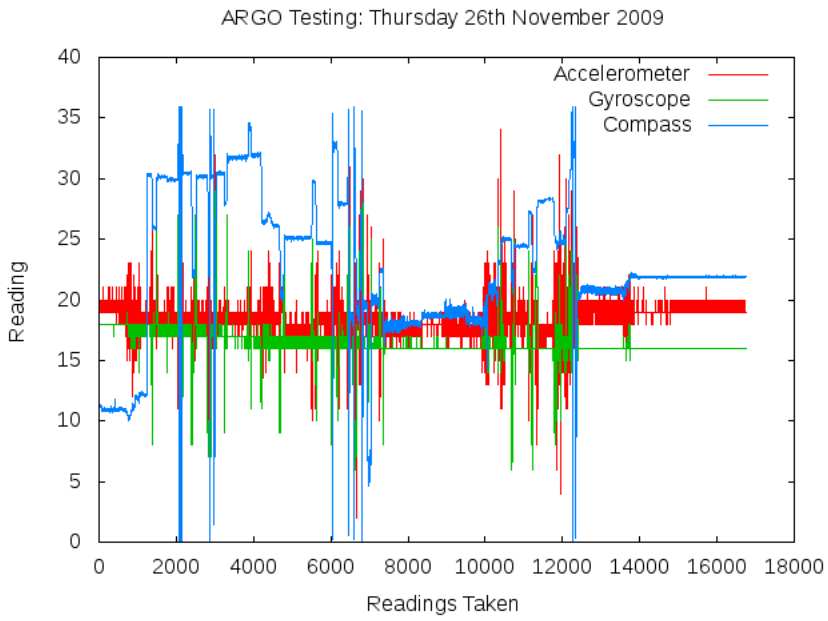


Figure 6.8: Test Result Data: 26th November 2009

to show on the same graph as the accelerometer and gyroscope. As such, a reading of 35 is actually 350 degrees etc.

Of interest in this graph are the values between 6,000 readings and 10,000 readings. During this period a brief telephone call took place and so the testing was halted for a few seconds. Further, after approximately 12,500 readings there were no more readings being taken. This was simply readings whilst the ARGO was stationary and the software was shut down. Both of these periods of time can be ignored, and that data will not be used for any analysis work.

Figure 6.9 shows a more detailed analysis of two turns during this test. It is clear from the compass data in the graphs that the first turn, between approximately 11,180 readings and 11,220 readings was a 45 degree turn from approximately 225 degrees from North to 270 degrees from North.

Having the compass data allows for significantly more actuate analysis of the data. In addition, it starts to show with some accuracy the point at which a turn truly begins. As seen from Figure 6.9 the accelerometer and gyroscope start to get readings far before the heading changes. It is suggested that this is caused by the build up of the throttle, and therefore the resulting vibrations, required before the vehicle truly begins to turn.

The results also show that even after the vehicle has finished a turn (that is the compass value is no longer varying) there are still residue vibrations shown by the accelerometer and gyroscope. For example, from approximately readings 11,130 to 11,150 the compass remains static yet the accelerometer is particularly active.

In addition, the graph also shows some interesting characteristics of the vehicle at the end of a turn that were not previously visible due to the lack of sufficient data. Specifically at approximately 11,130 to 11,140 readings the compass is static (suggesting the end of the turn) but the gyroscope records significant accelerations in a direction opposite to the turn direction. It is suggested that this is where the vehicle truly came to a stop, and could even

be characteristics of the wheels flexing.

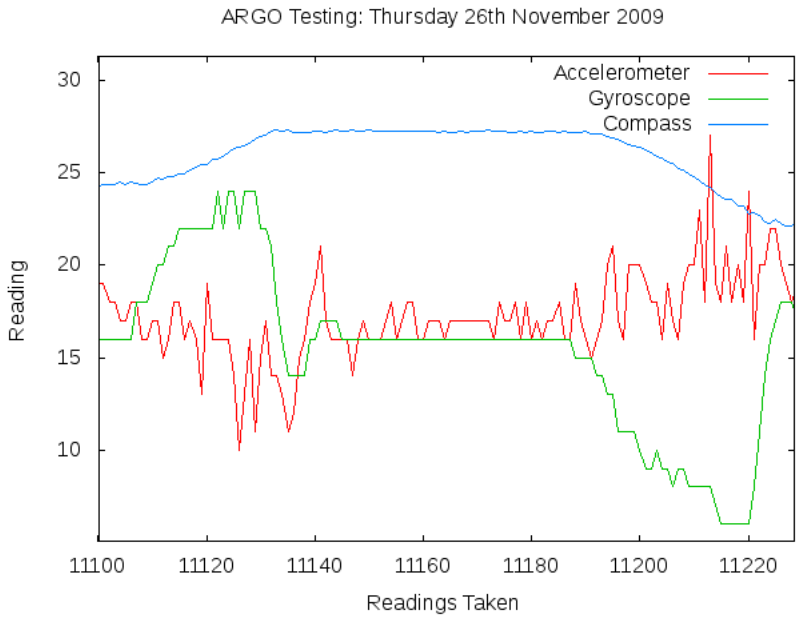


Figure 6.9: Detailed Test Result Data: 26th November 2009

6.3 Planning

As required in SCRUM during the pregame stage a basic plan has been developed that uses the high-level requirements identified previously (Chapter 2).

6.3.1 The Project Backlog

The first stage in the planning process is to identify a full list of requirements that will be put into the project backlog based on the functional requirements already identified (Chapter 2) and the investigative work already conducted. During the project, the backlog may be modified after each sprint to include more tasks, or to remove tasks that are no longer necessary due to the agile nature of SCRUM.

1. Control Software
   (a) Retrieve Sensor Data
      i. Get accelerometer readings.
      ii. Get gyroscope readings.
      iii. Get compass heading.
      iv. Retrieve GPS position information.
   (b) Vehicle Control
      i. Servo and actuator hardware control code.
      ii. Set engine speed/throttle position.

iii. Set handlebar position.

(c) Turning and Maneuvering

i. Turn by angle algorithm.
ii. Turn to angle algorithm.

(d) Teleoperation Functionality

i. Retrieve and parse user input.
ii. Increase throttle.
iii. Set steering left/right.
iv. Emergency Stop.
v. Get and display compass heading.
vi. Start high-level example algorithms.

(e) High-Level Example Algorithm

i. Turn to various angles, 90, 180, 270, 360 etc.
ii. Turn by various angles, 45, 90, 180 etc.

2. Simulator Software

(a) Emulation of Vehicle

i. Get compass heading.
ii. Servo and actuator hardware simulation.
iii. Set engine speed/throttle position.
iv. Set handlebar position.

(b) Emulation of Low Level Control Algorithms

i. Turn by angle algorithm.
ii. Turn to angle algorithm.

(c) Emulation of Teleoperation Mode

i. Retrieve and parse user input.
ii. Increase throttle.
iii. Set steering left/right.
iv. Emergency stop.
v. Get and display compass heading.
vi. Start high-level example algorithms.

(d) Simulator Framework

i. Loading of maps.
ii. Dynamic surface types.

(e) Graphical Interface

i. Display of the map.
ii. Continually updating display of the ARGO and its current position on the map.
iii. Display of sensor data and the current status of the ARGO, such as the coordinates, heading, speed, throttle etc.

6.3.2 Current Situation

Thanks to the investigative work already completed a significant amount of work has already been completed towards a number of the project objectives, including a brief analysis of the characteristics of the ARGO under manual control.

In addition, the initial investigative work has included the development of very primitive code to gather some initial sensor readings. Whilst this code will be "thrown away" in favor of higher quality production level code it has still been a useful exercise in identifying how to interact with a significant proportion of the hardware. This in turn has helped to minimise a number of the risks identified (Chapter 4). Further, it has assisted greatly with the identification of tasks for the project backlog, and the ordering in which those tasks should be developed.

Based on the information available, the ultimate project deadlines set by the university, the spike work already completed and the project backlog it is suggested that five sprints will be required for the development of the simulator and control system. Further, each sprint is expected to last approximately 2 weeks.

However, given the methodology being used is an agile methodology it is acceptable that the requirements and timings could change during the project, and as such the information following in this section should be considered an initial estimation only.

6.3.3 Sprint 1

The first sprint is planned to start on the 23rd November 2009 and last two weeks. During this time the commands to complete the following tasks are expected to be completed from the backlog:

- 1.a.i Get accelerometer readings.
- 1.a.ii Get gyroscope readings.
- 1.a.iii Get compass heading.
- 1.a.iv Retrieve GPS position information.
- 1.b.i Servo and actuator hardware control code.
- 1.d.i Retrieve and parse user input.
- 2.a.i Get compass heading.
- 2.a.ii Servo and actuator hardware simulation.
- 2.c.i Retrieve and parse user input.

Further, during this sprint work is also expected to take place installing motors and servos into the ARGO.

6.3.4 Sprint 2

The second sprint is planned to start on the 7th December 2009 and last two weeks. During this sprint, the following tasks from the backlog should be completed:

- 1.b.ii Set engine speed/throttle position.
- 1.b.iii Set handlebar position.
- 1.d.ii Increase throttle.
- 1.d.iii Set steering left/right.
- 1.d.iv Emergency stop.
- 1.d.v Get and display compass heading.
- 2.a.iii Set engine speed/throttle position.
- 2.a.iv Set handlebar position.
- 2.c.ii Increase throttle.
- 2.c.iii Set steering left/right.
- 2.c.iv Emergency stop.
- 2.c.v Get and display compass heading.

6.3.5 Sprint 3

The third sprint of the project is expected to start on the 4th January 2010 and should last for two weeks. Although the previous sprint finishes a few weeks before this the author anticipated that during the Christmas period little work is likely to be completed.

During this sprint, the following tasks from the sprint backlog are expected to be completed:

- 1.c.i Turn by angle algorithm.
- 1.d.vi Start high-level example algorithm.
- 1.e.i Turn by various angles, 45, 90, 180 etc.
- 2.d.i Loading map files.
- 2.d.ii Dynamic surface types.
- 2.e.i Display of the map.

6.3.6 Sprint 4

The fourth sprint is expected to start on the 18th January 2010 and again should run for two weeks. The start date of this sprint may need to be moved forward by one or two weeks due to exams during the end of January. During this sprint, the following tasks from the sprint backlog are expected to be completed:

- 1.c.ii Turn to angle algorithm.
- 1.e.ii Turn to various angles, 90, 180, 270, 360 etc.

- 2.b.i Turn by angle algorithm.
- 2.b.ii Turn to angle algorithm.
- 2.c.vi Start high-level example algorithms.

6.3.7 Sprint 5

This sprint is the final sprint and is expected to start on the 1st February 2010, though it may start later due to possible collisions with sprint 4 and the exam period. This sprint is expected to last two weeks, during which the following tasks in the backlog are expected to be completed:

- 2.e.ii Continually updating display of the ARGO and its current position on the map.
- 2.e.iii Display of sensor data and the current status of the ARGO, such as the coordinates, heading, speed, throttle etc.

6.3.8 Endgame

After all sprints are completed the endgame begins. Whilst normally in SCRUM testing would take place during this period, it has been decided that testing should take place at the end of every sprint instead. As such the end game will be used primarily for data analysis towards surface type detected. The endgame is expected to begin on the 15th February 2010 and will last at a maximum of 4 weeks (22nd March 2010).

At the end of the endgame the project should be completed, and fully functional. All testing and data analysis should also be complete. This will then leave approximately one and a half months for the completion of the project report.

6.4 High-Level Design

Although in SCRUM the design of the project is supposed to evolve during the course of the project and during each sprint, a brief overview design is suggested, as detailed in the development methodology (Chapter 5). In this section there is a brief overview of the current design for the simulator software and for the ARGO control software.

6.4.1 ARGO Control Software Design

Figure 6.10 shows a high-level overview of the current design for the ARGO control system.

The diagram shows a number of separate components and their communication links such as the I2C bus or a standard RS232 terminal connection. The two main components (the Gumstix and PIC) are briefly discussed below.

Gumstix

The Gumstix is responsible for the overall control of the system. It will have software running on it that will allow high-level control software to issue the various commands defined in the requirements analysis such as 'turn by angle' etc. In addition, it will be responsible for retrieving and processing all the sensor data from the other components. It has a direct serial link to the GPS module, a serial link to the PIC and a I2C link to the compass.

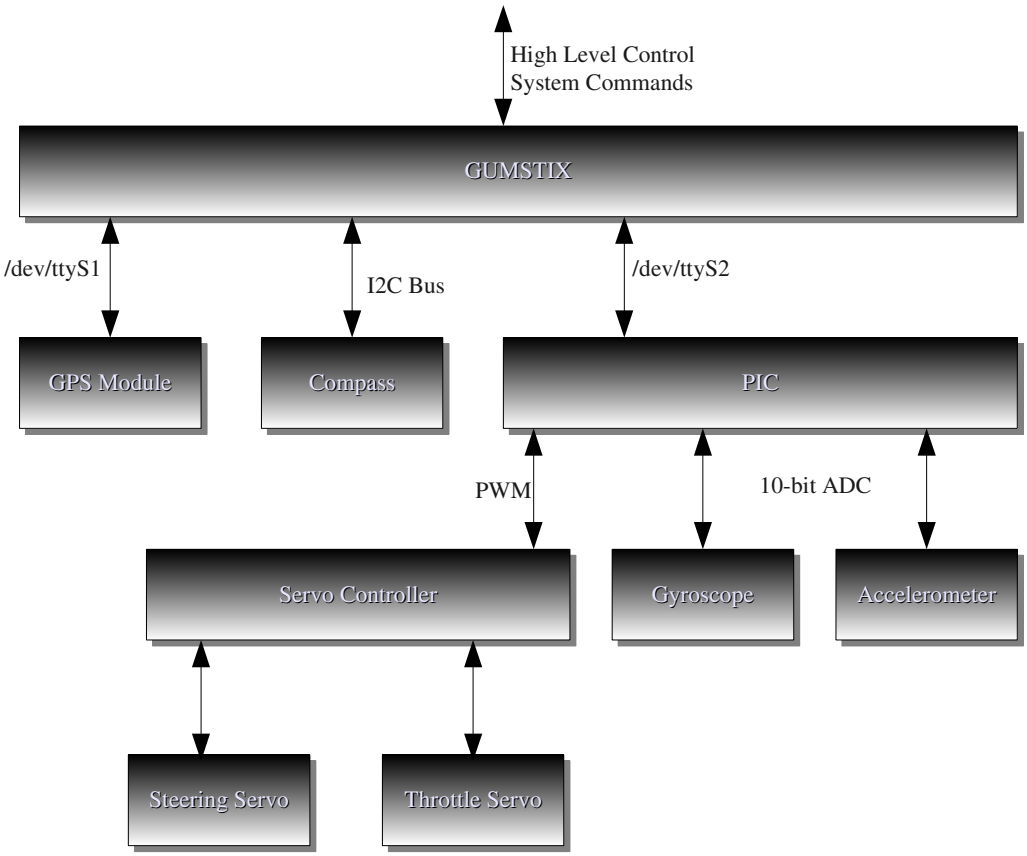


Figure 6.10: Overview of the ARGO Control System Design

PIC

The PIC is responsible for receiving commands from the Gumstix and then acting on those commands. The exact commands required will be identified at the appropriate spints, but are expected to be simple textual commands such as 'getGyro', 'getAccel', or 'moveServo' etc.

6.4.2 Simulator Software Design

As with the control software, there is a very brief overview of the initial high-level design for the simulator software.

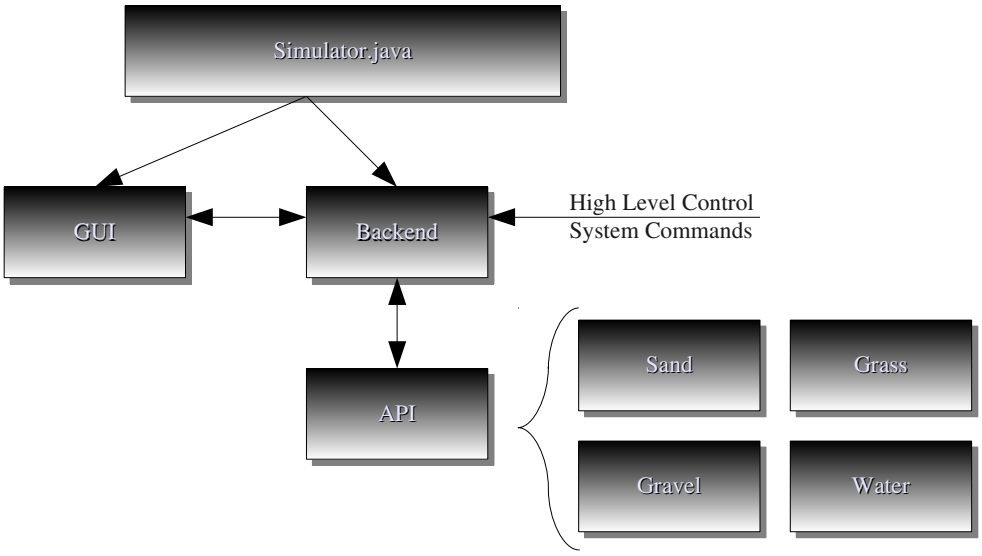


Figure 6.11: Overview of the Simulator Design

In this design there are three main aspects which are important and are discussed below.

Graphical User Interface (GUI)

This block represents the graphical user interface to the simulator. The GUI will communicate with the backend system to get information regarding where the ARGO is currently positioned etc. It is ultimately responsible for the drawing of the map and the 2D representation of the ARGO on the map.

As with all good design, the GUI should have no functional code, leaving all major processing to the backend classes. This way, should a different user interface be desired at a late point it can easily be added without major code modifications. Various design patterns such as Model View Controller (MVC) will be considered in the appropriate sprint for this purpose.

Backend

The backend block is the main bulk of the simulator. It is responsible for loading in the map file, displaying the teleoperation menu and simulating the command interface that the physical ARGO has. It will also be responsible for emulating the ARGO (sensor readings, heading, throttle, response times etc) and also for all the processing required for the movements of the ARGO when specific commands are received such as, for example 'setThrottle' or 'setHandlebarPosition' etc.

API

The application programming interface (API) block is used so that scientists and developers can easily write new surface types for the simulator. This means that if a new use for the ARGO is found that requires it to operate on a surface not previous accounted for (for example, the Mars surface) a new surface type can quickly and easily be developed and compiled and then loaded as a plug-in into the simulator.

It is intended that the API should be fully distributable without the need for the whole simulator package and that Java dynamic class loading and method invocation will be used to call the custom built surface types.

Chapter 7

Sprint 1

"If you think education is expensive, try ignorance."

Derek Bok

7.1 Sprint Planning

Being the first sprint of the project it started as planned on the 23rd November 2009. During the initial planning in the pre-game phase it was decided that the following features from the product backlog would be completed:

- 1.a.i Get accelerometer readings.
- 1.a.ii Get gyroscope readings.
- 1.a.iii Get compass heading.
- 1.a.iv Retrieve GPS position information.
- 1.b.i Servo and actuator hardware control code.
- 1.d.i Retrieve and parse user input.
- 2.a.i Get compass heading.
- 2.a.ii Servo and actuator hardware simulation.
- 2.c.i Retrieve and parse user input.

Further, being the first sprint there were no additional requirements or features left in the backlog from previous sprints, or additional features added to the backlog. However, it was anticipated that a number of additional design decisions would need to be made during this sprint such as precisely what hardware would be used, what communications protocols would be used between the differing hardware platforms etc.

Given the spike work already completed and the nature of the features in the backlog, this first sprint was expected to be relatively trivial allowing time for hardware components (such as actuators, servos, PICs etc) to arrive and be installed into the ARGO.

It was decided that the first half of the first week in the sprint should be primarily spent doing a large section of the design work. The next week would then involve the development of functions 1.a.i, 1.a.ii, 1.a.iii, 1.a.iv, 1.b.i, and 1.d.i. Finally the last half of the second week was to be spent writing the simulator functions 2.a.i, 2.a.ii and 2.c.i.

7.2 Design Decisions

This section covers the major design decisions that took place during this sprint.

7.2.1 Hardware Design

It was decided that due to the limited time available for the project, the experience already gained with the teaching boards during the pre-game phase and the wide range of sensors already available on the boards including GPS, a compass and a 3 axis accelerometer, these would be perfect for the development of the main control system. This immediately helped to simplify the design of the hardware for the final vehicle.

Further, it also meant that any modifications that would be needed (such as the addition of gyroscopes, servos and actuators) would have to be easily removable and modular. This had the added advantage that if the board failed at any point it would be trivial to replace it with a new one simply by installing the software and firmware and finally by connecting any external devices.

In addition to the sensors and processing hardware choices, it was also important to decide upon the most appropriate method of controlling the engine in the ARGO and the steering.

Steering

The steering system on the ARGO operates via handlebars which apply brake pads to a set of discs. These discs are directly connected via chains and sprockets to the wheels on either side of the vehicle. When the handlebars are moved, for example to a fully left position, the brakes on the left hand side are applied and the vehicle skids left.

The simplest method of modifying this system for computer control would involve moving the handlebars using an appropriate linear actuator. Unfortunately, whilst simple and low cost, the response times of such a liner actuator are expected to be within the region of two seconds to go from a full lock left to a full lock right. Further, given the nature of the system the effects of steering are only felt when the handlebars are at full lock.

The result of such a system would mean that it would be incredibly difficult to change the heading of the ARGO whilst moving. The control system would have to bring the vehicle to a full stop, apply the steering and then continue moving in the new direction.

The alternatives are significantly more complicated and would require the direct modification of the braking system such that each disc brake operates independently via separate actuators. This would have a number of advantages, in particular allowing for fast response steering allowing the vehicle to change direction whilst moving and full vehicle braking by applying both left and right breaks at the same time. However, such modifications would prevent the continued manual operation of the vehicle via the handlebars.

Given this information, whilst in the long term or for future projects it would be an interesting progression to attempt direct manipulation of the brakes, it has been decided that the simpler option of installing a liner actuator directly onto the handlebars is the most simple choice. Specifically, a LINAK LA12[1] actuator has been chosen for this purpose. Finally, in order to accept pulse-width-modulation, a simple motor controller will be used as the interface between the control system and the liner actuator.

[1]Linak website: `http://www.linak.com/Products/?id2=2`, Last accessed: 16th April 2010

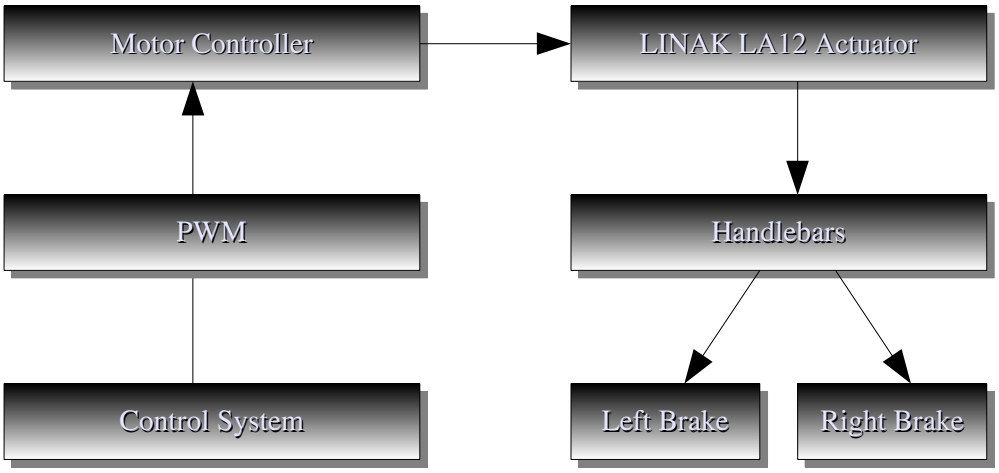


Figure 7.1: Steering Hardware Design

Throttle Control

When driven manually the throttle is adjusted via a grip-and-twist adjuster on the handlebars. This in turn adjusts a centrifugal governor. This then maintains appropriate throttle levels at the carburetter. Unfortunately, simply attaching a servo to the governor would significantly slow the response times of changing the throttle via a control system. For this reason it was decided that the governor should be completely disconnected and a simple servo would be placed directly onto the carburetter in its place.

This provides a number of advantages. First, it provided significantly faster response times from the engine when adjusting throttle values, something which is going to be vital for the development of the control system and secondly it allows for much greater control of the actual throttle value. Instead of allowing the governor to decide on the actual throttle value, the control system will be in complete control.

However, there are a number of disadvantages also. First, without the governor attached it is highly unlikely that the engine will run at peak efficiency and as such the engine may need more regular servicing, and further it may not provide as much power as the manufacturers anticipated. Secondly, it removes the possibility for the vehicle to be manually driven without re-attaching the governor. However, as this is an extremely trivial job which takes approximately two minutes this is not considered to be a major disadvantage. Further, with the teleoperation mode this is also not anticipated to be needed.

Steering and Throttle Control Overview

Having decided that the use of a LINAK LA12 for the steering and a simple servo for the throttle the final decision was how to actually control these devices. There were a number of options for the actual hardware/software interaction.

The first option was to use direct pulse-width-modulation via the PIC. This would have meant having a control loop or timer interrupt driven process on the PIC directly responsible for sending pulses to the throttle servo and motor controller for the steering.

The second option was to use an off the shelf hardware Pololu 16-Servo Controller board[2] which is specifically designed to take position information via a serial line and then performs constant pulse-width-modulation. This Pololu board would need to be run via the Gumstix rather than the PIC via a USB serial driver.

It was decided that the use of the Pololu board would be the most appropriate in order to get a clear division of labour in terms of the control system. The PIC will be solely responsible for retrieving sensor data, whilst the Gumstix will be responsible for the processing of the data and controlling the vehicle via the hardware components (Figure 7.2).

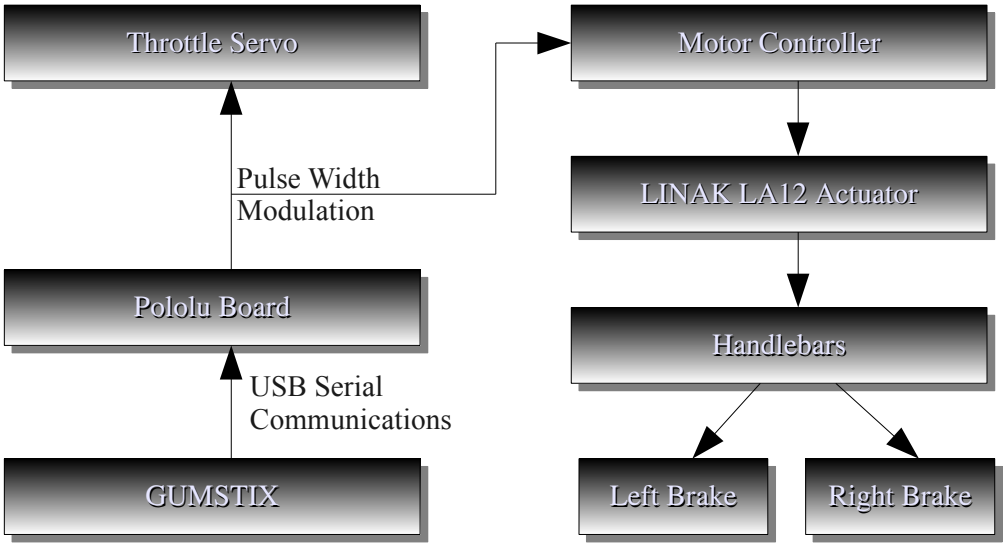


Figure 7.2: Steering and Throttle Hardware Design Overview

Safety Hardware

Clearly the autonomous operation of any vehicle poses risks to people in the near vicinity and a petroleum powered vehicle is no exception to this. It was identified during the

[2]Pololu Board website: `http://www.pololu.com/catalog/product/390`, Last accessed: 16th April 2010

risk analysis that safety would be of vital important during this project and a number of recommendations were made including always having human operators in the vehicle able to shut down power to the vehicle. Further, it was also suggested that the use of an industrial grade emergency shutoff system should be installed. As such this system will be wired directly to the ignition system of the ARGO. In the even of an emergency pressing the red stop button on the remote control will instantly cut power to the engine, preventing damage to humans and property.

Current Hardware Design Overview

Having made a number of decisions regarding the hardware platform to be used a new improved hardware design (Figure 7.3) is shown below. This design is in keeping with the original design during the pre-game phase but also incorporates the additional hardware design decisions made during the first sprint.

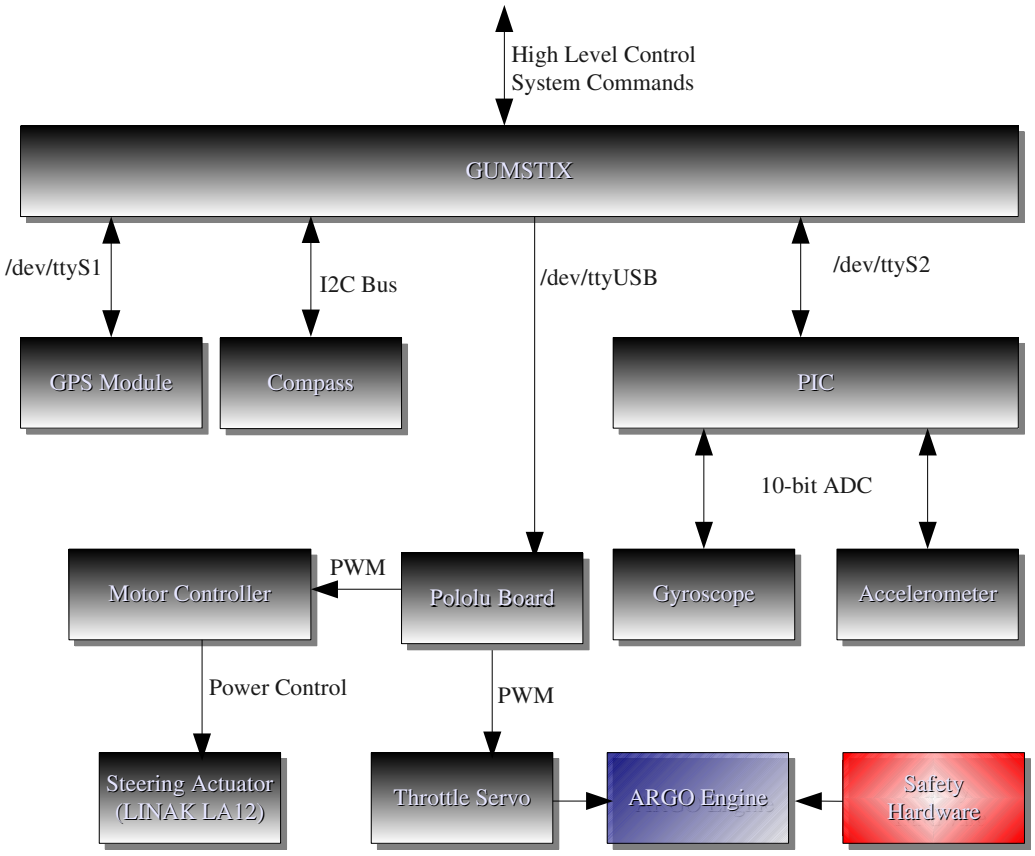


Figure 7.3: Sprint 1: Final Hardware Design Overview

7.2.2 PIC Firmware Design

As already discussed during the hardware design it is vital that there is a clear division of labour between the different sections of the control system. This is important as it helps to lead towards a clear and concise modular system which will significantly help towards further safety and ensuring that any real-time constraints are met.

Given the nature of the hardware design it is inevitable that the PIC will have to be responsible for the retrieval of data from sensors. However, as the PIC has no connections to any other hardware, it will have no other responsibilities. As such a number of possibilities are available for the development of the PIC system.

The first option is that the PIC has a external interface to retrieve requests for data from the Gumstix. A simple protocol would then be developed that would operate as shown below (Figure 7.4).

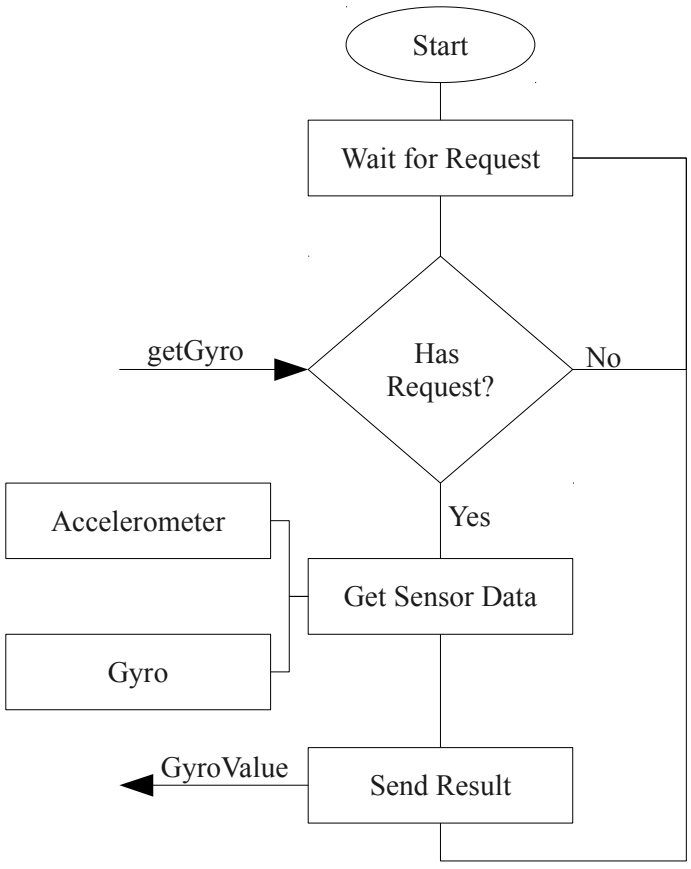


Figure 7.4: Possible PIC algorithm where requests for sensor data are received.

This design has one major advantage - that only the data needed is ever collected and sent. This means that time is never 'wasted' collecting sensor data that will not be used. However, there are significant overheads with this design. If all sensor data is required it would mean a number of repeated requests for every sensor. This would add significant processing time that is undesirable.

A modified version of this design (Figure 7.5) could be used that has a significantly

more simplified algorithm. In this instance, a request would be processed, not for individual sensors, but for all the sensor data at once.

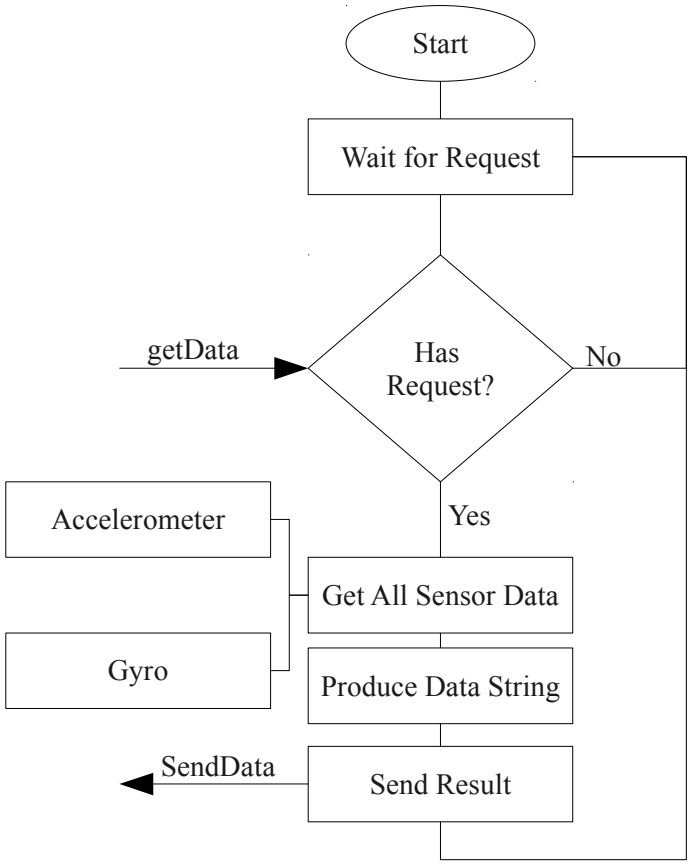


Figure 7.5: Possible PIC algorithm where a request for all sensor data is received.

Whilst this algorithm is more sensible there are still issues. Firstly, because of the haphazard nature of requests it would prevent the PIC being usable for any other operations in the future. Secondly, as all data is always being returned anyway, there seems no point in having the added complications of processing data requests. Further, having a request data, send data protocol provides more opportunities for errors. If the Gumstix were to send the wrong request and then wait for an answer, the PIC may never respond due to invalid requests etc.

As such, a much simpler solution will be used in the final system. The PIC will simply sit in a tight loop continually getting sensor data, producing data strings and sending them via the serial line. It will then be the responsibility of the Gumstix to simply read the serial line for data when it needs the information. This also has the advantage that if at a later date other devices also need this sensor data, they could also simply read the serial line, removing any possible future concurrency issues regarding requests to the PIC for information.

A very simple data string (Figure 7.6) will be used for transmission via the serial line. The data string will be precisely 15 digits long. First will be the accelerometer values and

finally the gyroscope value. All values will be precisely 3 digits long due to the 8-bit limited value from the analogue to digital converter, and all values will be comma separated.

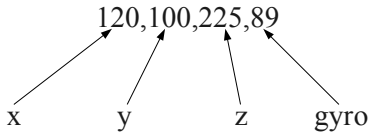


Figure 7.6: Transmitted PIC Data String

7.2.3 Gumstix Software Design

Although the C programming language is not object-orientated it is clearly important that the various tasks that the control system needs to perform should be divided into separate files. This will both help with the clarity of the code and the maintainability, for example, it would be clear at a later date that the retrieval and parsing of data from the PIC would be contained with all functions in the single pic.c file. As such, this section uses standard UML Class Diagram notation, where each class actually represents either a .c file, or a structure as appropriate.

pic.c and pic_data_t

All data from the PIC will be stored in a single structure. This structure is depicted in Figure 7.7. All the values are of type uint8_t due to the fact that the data from the PIC is always 8 bits in length.

There will be a need for three functions to read the data from the PIC. The first of these three functions, *open_pic(rate, char)*, will be responsible for opening a serial link with the PIC so that data can be read. The baud rate of the PIC will be set at 19,200bps. The serial line the PIC is installed on is located at /dev/ttyS2. The *close_pic()* function has a similar requirement, it must simply close the serial line.

Finally, the *get_pic_data(pdt_ptr)* is responsible for reading a single line from the PIC, parsing it into the individual components (i.e. accelerometer readings, gyroscope etc) and then populating the pic_data_t that was passed in as a pointer.

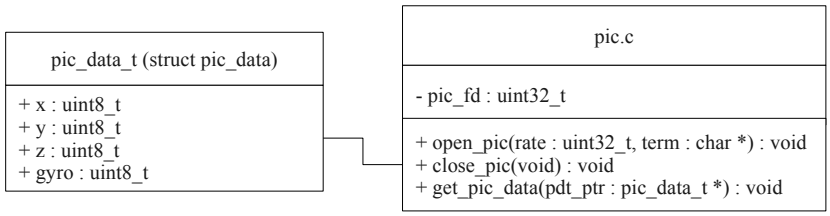


Figure 7.7: Gumstix Software PIC UML Diagram

gps.c and gps_data_t

The GPS module being used is connected to a serial line with the Gumstix. It has a relatively simple protocol for communications, as shown in Figure 7.8.

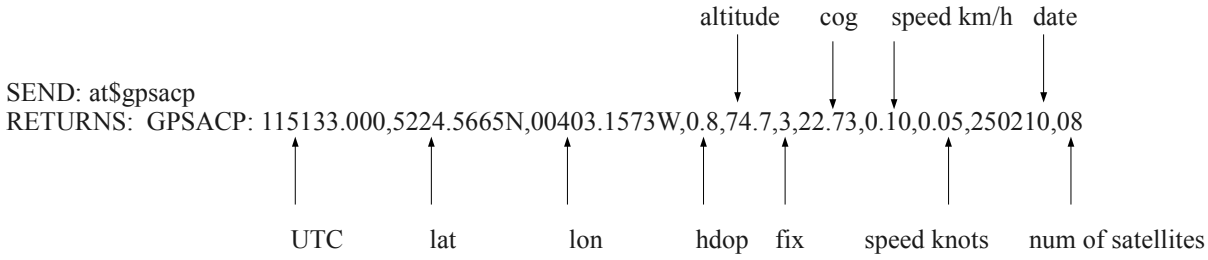


Figure 7.8: GPS Protocol

Each of the values in the response string will be stored in a data structure called gps_data_t, as shown in Figure 7.9. In addition there will be four functions required in order to communicate with the GPS module, retrieve and process the resulting GPS data string.

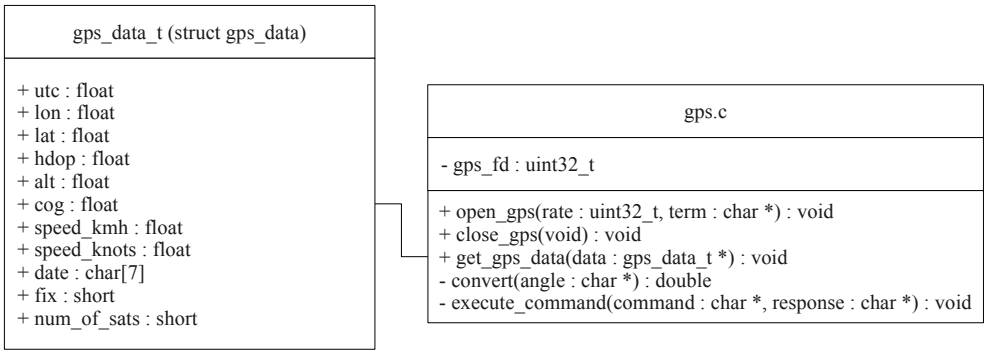


Figure 7.9: Gumstix Software GPS UML Diagram

The first two functions have almost identical requirements to those in the PIC code, *open_gps(rate, char)* and *close_gps()* are both responsible for opening and closing the serial line to the GPS module. The baud rate for the GPS module is higher than the PIC at 115,200bps as is located at /dev/ttyS1. The *get_gps_data(data)* function is used to retrieve the current GPS information from the GPS module and then populate the gps_data_t structure pointed to by the data pointer.

In addition to these three 'public' functions, two more will be used. The *convert(angle)* function will be used to convert the longitude and latitude from the Degrees Minutes Seconds format from the GPS module to decimal degrees. This will hopefully make any future work easier as decimal degrees are more appropriate for distance calculations, waypoint navigation etc.

Finally, the *execute_command(command, response)* function is used to directly execute the AT commands on the GPS unit and return the response in the response buffer.

compass.c

The compass uses the I2C bus for communications between itself and the Gumstix. A library is already available that both deals with the low level requirements and implements the I2C protocol allowing for relatively simple communications with I2C devices. The library[3] is part of the standard gumstix-buildroot and is licensed under the GNU Version 2.

Given the use of the library, all that will be required for compass communications is the appropriate calls to the I2C library and three functions are sufficient to do this. The *open_compass(compass_address)* function is responsible for initialising the I2C bus for communications with the compass at the given address. The *close_compass()* function releases the I2C bus. Both of these functions are called whenever a read needs to take place. The *get_compass_data()* function is responsible for making the appropriate calls to the I2C library to retrieve the current compass heading.

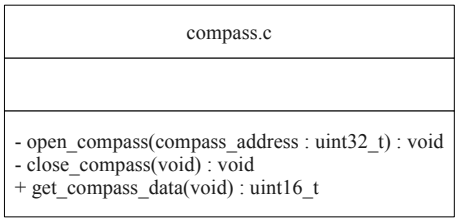


Figure 7.10: Gumstix Software Compass UML Diagram

servo.c

The Pololu board is to be used for all control of servos, motors and actuators in the project. It is capable of continually generating pulse-width-modulation for 16 separate peripheries. The Pololu board also uses a simple USB serial line for communications with the Gumstix. The functions *open_servo(rate, term)* and *close_servo()* are responsible for opening the serial line (located at /dev/ttyUSB0) with a baud rate of 9,600 (the maximum available for the Pololu board) and closing it.

Once a serial line is open, there are two different protocols available for communicating with the Pololu board. The simplest mode, Mini SSC II Mode, provides precisely the required features for this project. As such this mode will be used rather than have the complications of the 'Pololu Mode' which are simply not necessary.

The Mini SSC II Mode protocol is very simple. It requires that three bytes are sent. The first byte is a start byte and this should always be 0xFF. The next byte is the servo number that the operation should take place on, this should be in the range 0x00 to 0x0F (0 - 15). The final byte is the new position of the servo, and this should be in the range 0x00 - 0xFE (0 - 254).

[3]I2C library available for download from: `http://svn.gumstix.com/gumstix-buildroot/branches/projects/robostix/gumstix/Common/`, Last accessed: 18th April 2010

In the ARGO control system, the *set_servo(servo, position)* will be responsible for formulating this arrangement of bytes, taking the servo number and the new position as parameters.

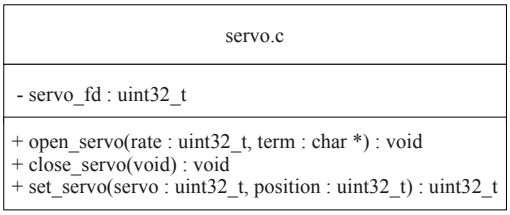


Figure 7.11: Gumstix Software Servo UML Diagram

argo.c

The final stage in the design of the control system, at least in terms of the tasks specified, involved the acquisition of user input. It was decided that this should this should simply take place in the main method.

It was further decided that all key presses should be processed directly by the program rather than having the terminal buffer the commands. As such a standard 'termio' structure will be needed in order to disable buffering or blocking I/O from the terminal. This will mean that all key presses are directly, and instantly, handled by the control system.

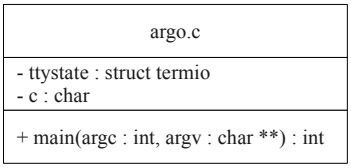


Figure 7.12: Gumstix Software Main ARGO Method UML Diagram

Final Gumstix Software Design

The final overview of the design for the Gumstix software to the end of this sprint is shown in Figure 7.13.

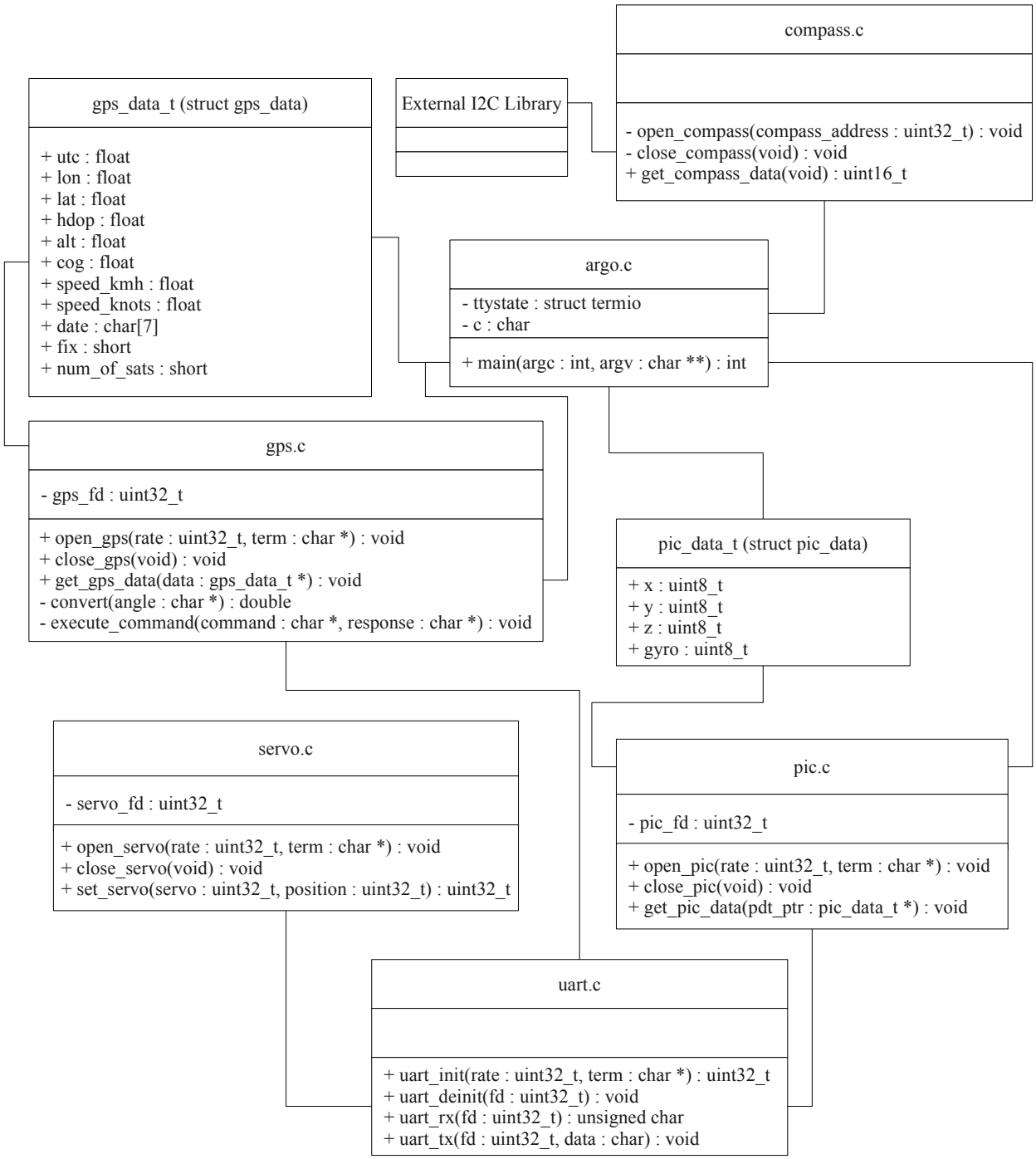


Figure 7.13: Gumstix Software UML Design

7.2.4 Simulator Software Design

The simulator software design mimics as closely as possible the design of the physical control system and is detailed in this section.

Packages Overview

The simulator will be split into three distinct packages. A brief overview of these three packages is given. The first package will contain all the class files which simulate the physical robotic platform. It will be called *uk.ac.aber.dcs.sem9060.backend*. The second package, *uk.ac.aber.dcs.sem9060.api*, will provide a simple API (Application Programming Interface) for developers who wish to produce surface types. The final package, *uk.ac.aber.dcs.sem9060.gui*, will hold a simple graphical user interface that will communicate with the backend for display purposes only.

Simulator.java

The main Simulator file will be responsible for creating an instance of the backed and passing control to it. It simply holds the main method.

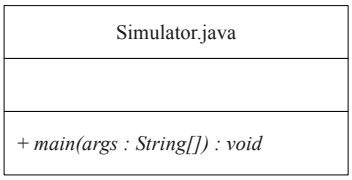


Figure 7.14: Simulator Software Simulator Class UML Diagram

Backend.java

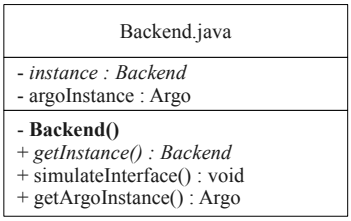


Figure 7.15: Simulator Software Backend Class UML Diagram

The backend of the simulator is to be responsible for human-simulator interaction by emulating the control interface of the ARGO, and for the direct manipulation of the simulated ARGO. For this reason, it is vital there there is only ever a single instance of the

backend. If two backend instances were ever to occur it could result in both of them attempting to make different control decisions on the ARGO at the same time. This would clearly be unacceptable.

For this reason it has been decided that the Backend class should use a singleton design pattern. As such, a static method *getInstance()* exists which is required to both create a instance of the Backend if one does not already exist, and then return to the caller the current instance. To prevent other classes from creating an instance of the Backend, the constructor is to be private.

The *simulateInterface()* method will display a menu and act like the 'argo.c' file in the physical implementation. However, it will be required to run in its own thread so that at a later date, commands can continue to be sent to the backend even whilst the front end graphical user interface is performing updates, or because the backend is simulating another control maneuver. Finally the *getArgoInstance()* method will simply act as a 'getter' in order for other parts of the control system, or the front end, to be able to get the current instance of the simulated Argo, for example for displaying its location on a map.

Argo.java

The final class for this sprint is the Argo class. It has the sole job of representing the physical characteristics of the physical Argo. This means that, at the moment, it simply holds a compass heading. However, as work progresses during the development this class is expected to increase in size to include coordinates and possibly other information such as the speed of the vehicle etc.

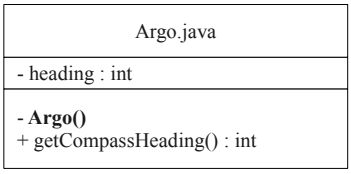


Figure 7.16: Simulator Software Argo Class UML Diagram

Final Simulator Software Design

An overview UML diagram of the final simulator software design for this sprint is shown in Figure 7.17. Further, all whiteboard drawings for the entire of this sprint relating to the design work can be found in Appendix A.

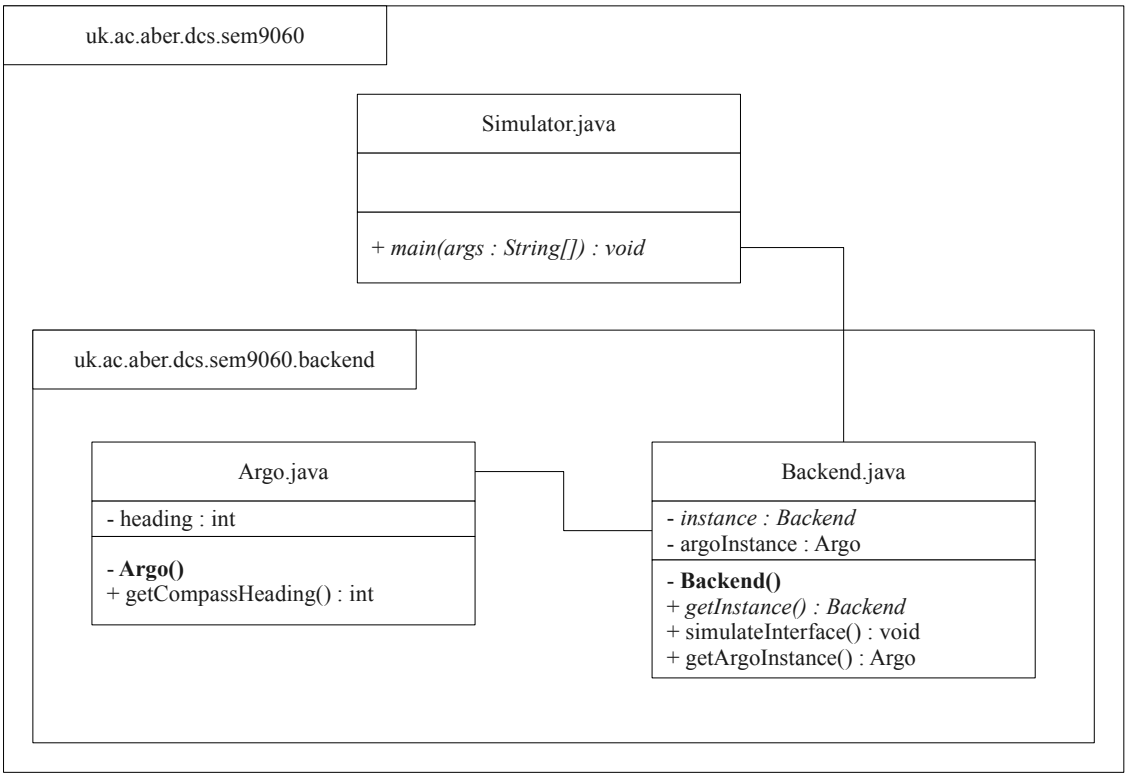


Figure 7.17: Simulator Software UML Design

7.3 Implementation

The actual implementation of the system was relatively simple thanks to a lot of the tasks having a very similar nature to the investigative work. However, there was one notable issue that occurred. The design for the protocol between the PIC and Gumstix proved to be unacceptable. Although the PIC was sending the correct data, because there was no synchronisation, the Gumstix often started reading a data string half way through transmission. With no identification of the start, or end, of a transmission it would often mean that the gyroscope value would get treated as a accelerometer value etc.

In order to get around this issue the protocol string between the Gumstix and PIC was modified such that it has a start and end string that must be matched. The code on the Gumstix was then modified to wait until the correct start sequence was identified from the PIC before reading any values.

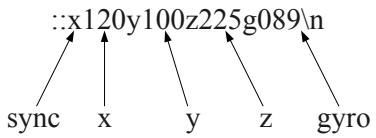


Figure 7.18: Revised PIC Data String

Adding this synchronisation information fixed all the issues that were being seen.

7.4 Testing

At the end of each sprint it is important to perform testing. Testing of features at the end of each sprint will help to prevent problems as the project progresses. Faults or errors caught early on in the development phase will significantly help in the reduction of code that must be re-written, time and effort. Further, showing that the functions at this low level are working correctly will mean that if there are faults later on it will be possible to confirm the faults are not in the lower level (by re-running the tests and comparing output). In addition, the testing framework will mean that if changes are required to the code already completed at a later date, it will be possible to directly check that the functionality has not been affected. Having said this, a lot of the functionality developed thus far is very hard to test. In spite of this a simple test strategy was devised and the results are indicated in the table (Figure 7.19).

Test	Feature	Expected	Result	Pass/Fail
P1	Read accelerometer.	Accelerometer values change as the hardware is Manipulated.	Changing values.	Pass
P2	Read gyro.	Gyro values change as the hardware is manipulated.	Changing values.	Pass
P3	Sent string from PIC is in correct format.	See design.	As per design.	Pass
G1	Read compass.	Display compass value.	265 degrees	Pass
G2	Read string from PIC is in correct format.	See design.	As per design.	Pass
G3	Move any servo.	Servo attached moves.	Servo moved.	Pass
S1	Read compass.	Value: 0 degrees	0 degrees	Pass

Figure 7.19: Sprint 1 Test Results, P: PIC, G: Gumstix, S: Simulator

7.5 Sprint Review

Despite being a relatively trivial sprint in terms of development work, a significant portion of time has been spent doing design work. In particular, a lot of the basic frame work for the physical control system has been completed meaning it will now be possible to progress to the other tasks in the product backlog.

During the sprint, the only notable issue was discussed during the coding section and related to the protocol used for communication between the PIC and Gumstix. However, this was sorted with minimal changes to the design and proved to be a trivial fix.

All tests that could be conducted have passed, and all features that were to be completed during this sprint were developed to an acceptable level and as such have been removed from the project backlog. Finally, according to the original plan, the project is on schedule.

Chapter 8

Sprint 2

"You will never find time for anything. If you want time, you must make it."

Charles Bixton

8.1 Sprint Planning

The second sprint started on time on the 7th December 2009. As there were no features uncompleted from the previous sprint, and no additional features added, the product backlog was as defined during the pre-game state.

- 1.b.ii Set engine speed/throttle position.
- 1.b.iii Set handlebar position.
- 1.d.ii Increase throttle.
- 1.d.iii Set steering left/right.
- 1.d.iv Emergency stop.
- 1.d.v Get and display compass heading.
- 2.a.iii Set engine speed/throttle position.
- 2.a.iv Set handlebar position.
- 2.c.ii Increase throttle.
- 2.c.iii Set steering left/right.
- 2.c.iv Emergency stop.
- 2.c.v Get and display compass heading.

8.2 Design Decisions

During this sprint significantly less design work was required compared to the first sprint. However, there were some changes to the current design, including a minor hardware design change. The details of these changes, and the details of the new components of the system are discussed in this section.

8.2.1 Hardware Design

Unknown in the first sprint, the actuator that will control the handlebars does not function in the same way as the standard servo to be used on the throttle. The motor controller sets the speed of the handlebars and the direction, not a position. It is then the requirement of software to read a potentiometer installed within the actuator to identify its position.

As a result of this a minor change needs to be made to the PIC hardware design, the potentiometer wire from the actuator will be fed into the analogue-to-digital converter as well as the accelerometer and gyroscope.

8.2.2 PIC Firmware Design

As a result of the hardware changes the PIC will now be responsible for sending the potentiometer value to the Gumstix in addition to the accelerometer and gyroscope. As such, the data string will need to be changed to 22 digits in length: the two start colons, the accelerometer values, the gyroscope value and finally the potentiometer value.

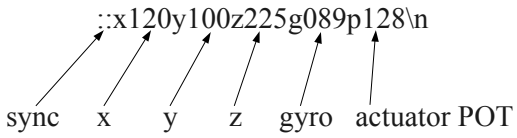


Figure 8.1: PIC Data String Including Actuator POT

8.2.3 Gumstix Software Design

This section reviews the design changes made to the Gumstix software in order to accommodate the new hardware design. It then discusses the additional design decisions made with regards to new features to be developed from the product backlog.

pic.c and pic_data_t

As a result of the changes regarding the potentiometer and the PIC firmware, there is a small change to the pic_data structure. It will now require a attribute to hold the current position of the handlebars, as shown in the updated UML diagram (Figure 8.2).

argo.c

Finally, in order to accommodate for the requirements 1.d.ii to 1.d.v the main function within the argo.c file will need to be modified to include a simple menu. This will require no design changes, but will require additional code within the main function. For simplicity, it has been decided that the menu should have the following options available.

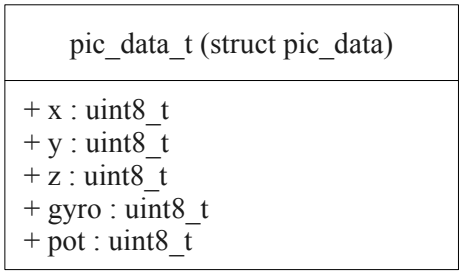


Figure 8.2: Gumstix Software pic_data_t UML Diagram

Option	Action
w	Increase Throttle
a	Handlebars To Left
s	Full Stop, Center Handlebars, Throttle to Minimum
d	Handlebars To Right
c	Get and Display Compass Reading

throttle.c

Given the current progress towards the control of servos from the previous sprint and the use of a Pololu board, the design for the control of the throttle is trivial.

Two functions are required, *init_throttle(motor)* which will be used to tell the control system the port number on the Pololu board that is connected to the throttle servo, and *set_throttle_position(speed)* which is to be responsible for setting the position of the throttle servo. It does this by simply using the *set_servo(servo, position)* function within the servo.c file.

Unfortunately, due to the way the throttle servo has been attached to the carburetor, the maximum throttle position is a smaller integer than the minimum throttle value. As such, in order to increase the throttle, and thus the speed of the vehicle, a smaller number must be passed in. To make this slightly easier, the maximum throttle position, `THROTTLE_MAX`, which is actually 1, and the minimum throttle position, `THROTTLE_MIN`, which is 81, are to be made available throughout the code base as `#define` variables.

steering.c

The steering for the ARGO will be controlled via the steering.c file, as depicted in Figure 8.4. Unlike with the throttle code, the steering is more complicated. When a desired position is received (from anything in the range `STEERING_LEFT` to `STEERING_RIGHT`) the software must turn the actuator on with an appropriate speed and direction, depending on the current position of the handlebars. Further, the software must continually check for the current position of the steering, until such a time as the desired position is achieved. At this point, the software must stop all movement of the steering. However, due to the potential response times from the steering, it may be the case that the steering overshoots the anticipated position, and as such the software must be able to reverse the direction of

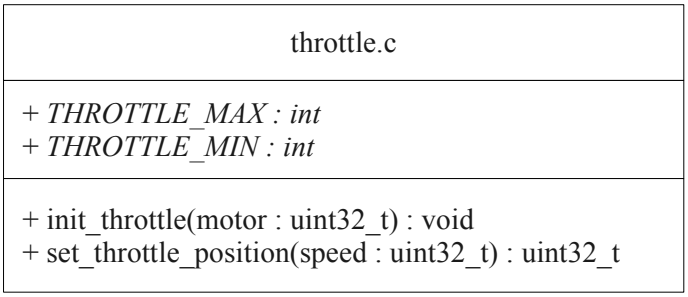


Figure 8.3: Gumstix Software Throttle UML Diagram

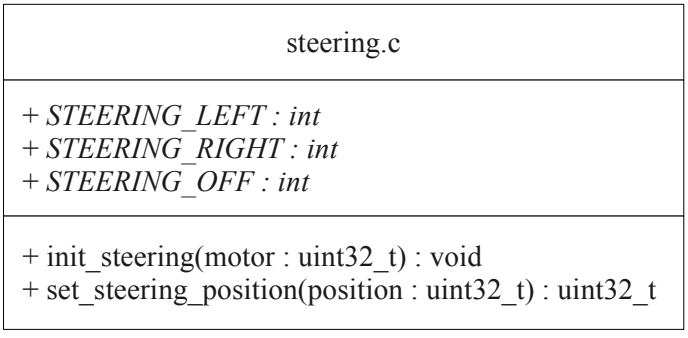


Figure 8.4: Gumstix Software Steering UML Diagram

travel at any point. In order to achieve this a simple algorithm, as shown in Figure 8.5, will be used.

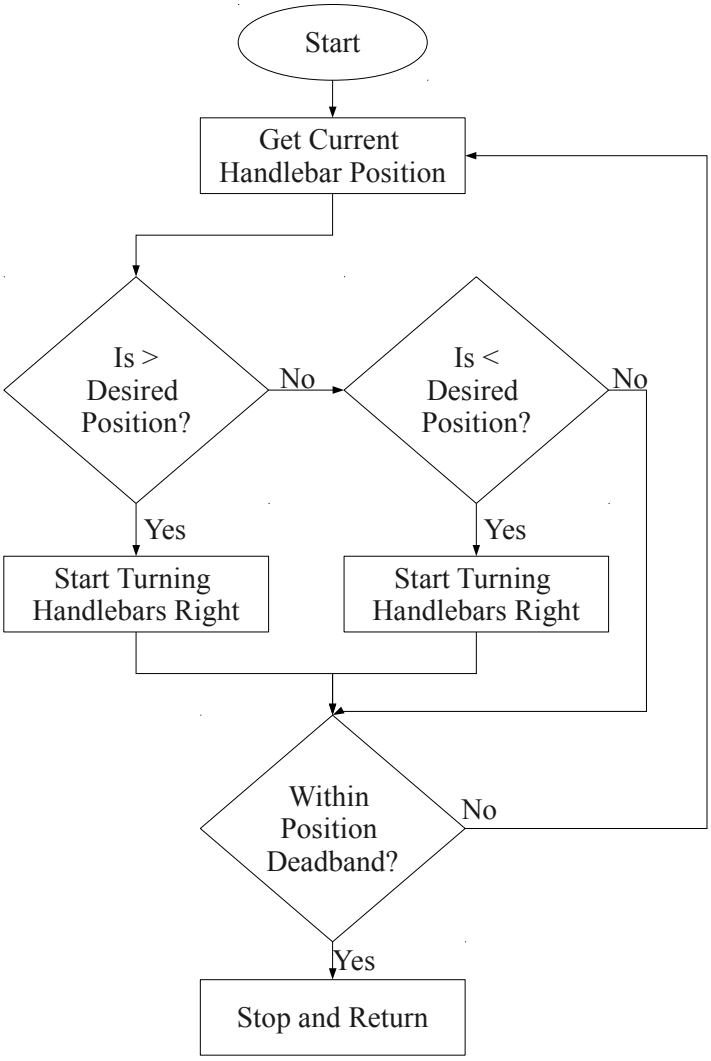


Figure 8.5: Steering Algorithm Flow Diagram

Final Gumstix Softwre Design

The final design of the Gumstix software to this point, including all modifications made during this sprint is shown in Figure 8.6.

8.2.4 Simulator Software Design

Only minimal design changes were required to implement features within the product backlog for the simulator. Those design changes are briefly discussed in this section.

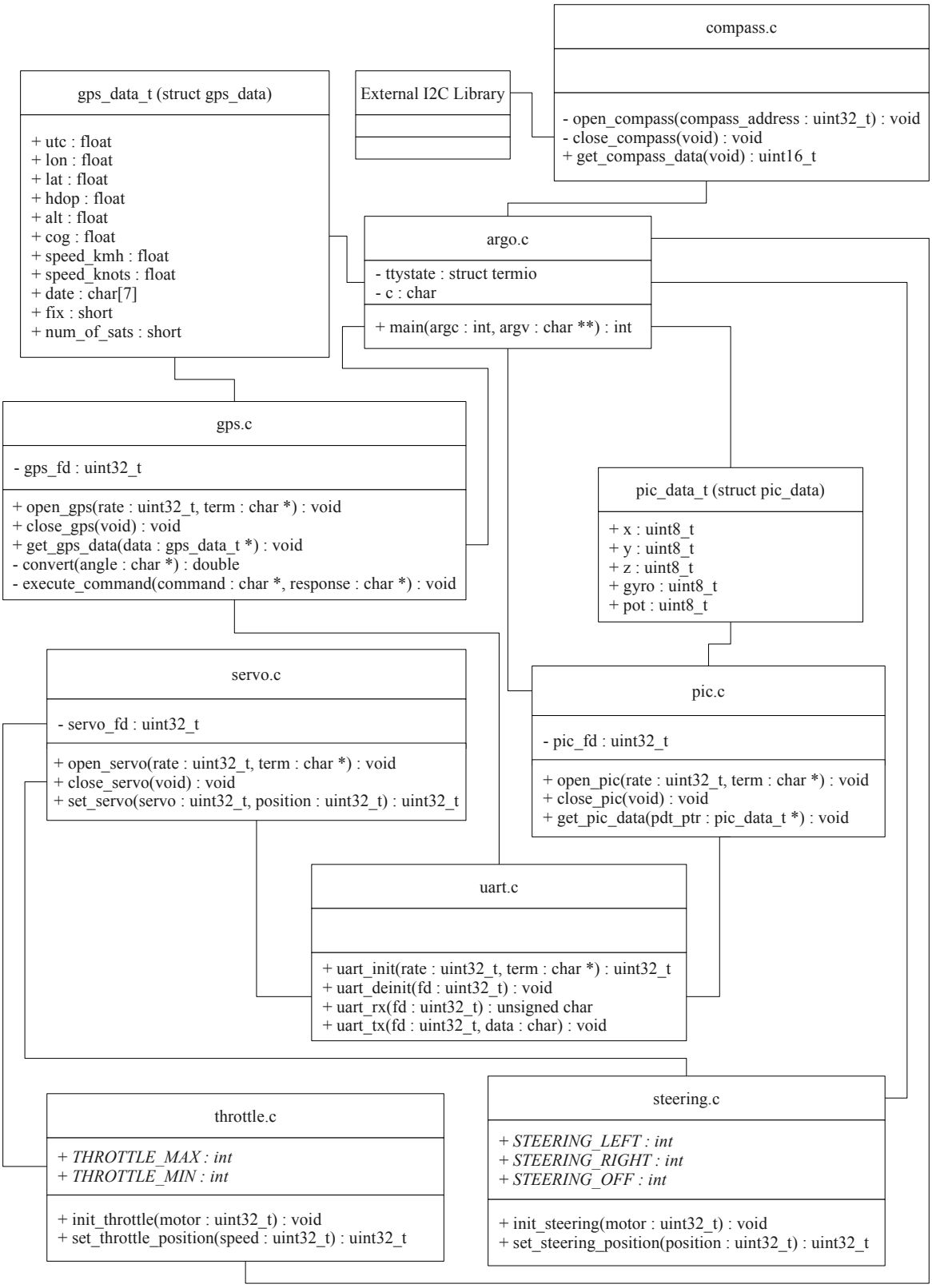


Figure 8.6: Gumstix Software UML Design

Argo.java

The main changes to the design are within the Argo class as shown in Figure 8.7.

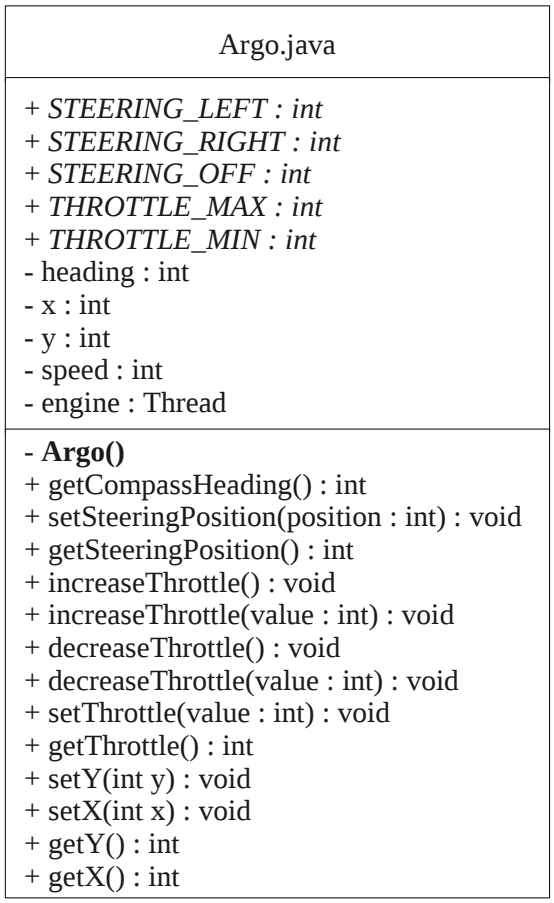


Figure 8.7: Simulator Software Revised Argo Class Diagram

A number of new parameters have been added, as have various methods. The function *increaseThrottle()* and *increaseThrottle(value)* will increase the throttle speed by one, or by any given value respectively. By the same token, the functions *decreaseThrottle()* and *decreaseThrottle(value)* will have the opposite effect, decreasing the throttle speed by one, or by the specified value. All of these methods will eventually use the *setThrottle(value)* function to physically manipulate the `speed` attribute.

The *setSteeringPosition(position)* method will set the position of the handlebars between `STEERING_LEFT` and `STEERING_RIGHT`. Any value out of this range will keep the steering at its current position, as is the functionality with the real ARGO. The function will delay for at least two seconds to simulate the real time the handlebars take to move on the physical ARGO.

The only other notable change is that a `engine` attribute has been created. This will simulate the actual engine in the ARGO. As such, when a call is made to *increaseThrottle()* for example, the `speed` value will be increased. However, this will not have an effect on the vehicle until the `engine` thread picks up the change after some appropriate time delay.

This is the expected functionality of the ARGO, simply increase the throttle does not have a instantaneous effect on the vehicle.

Whilst at this stage the time delay and the engine response will be a static value, eventually it is desired that these values will be retrieved from the specific surface type that the ARGO is positioned over. It is via this method that it is hoped to have the simulator react differently to differing surfaces.

The use of a thread in this situation is further required due to the way in which the control system and humans interact with the vehicle. The ARGO may be performing an operation such as turning. At this point the engine will have a certain throttle speed and so the engine thread will be turning the ARGO at an appropriate speed for that throttle value. However, it is clear that eventually the control system will need to tell the ARGO to stop moving. If threads were not used in this situation it would mean that the implementation of performing a turn would also have to be in the engine thread! At this point all object-orientation and code coherency would be lost!

Backend.java

Although there are no design changes, a menu identical to that used in the physical ARGO, with the operations increase throttle, stop, handlebars left, handlebars right and get compass value will be added in order to emulate the physical interface as closely as possible.

Final Simulator Design Overview

The new complete UML diagram for the simulator software design is shown in Figure 8.8.

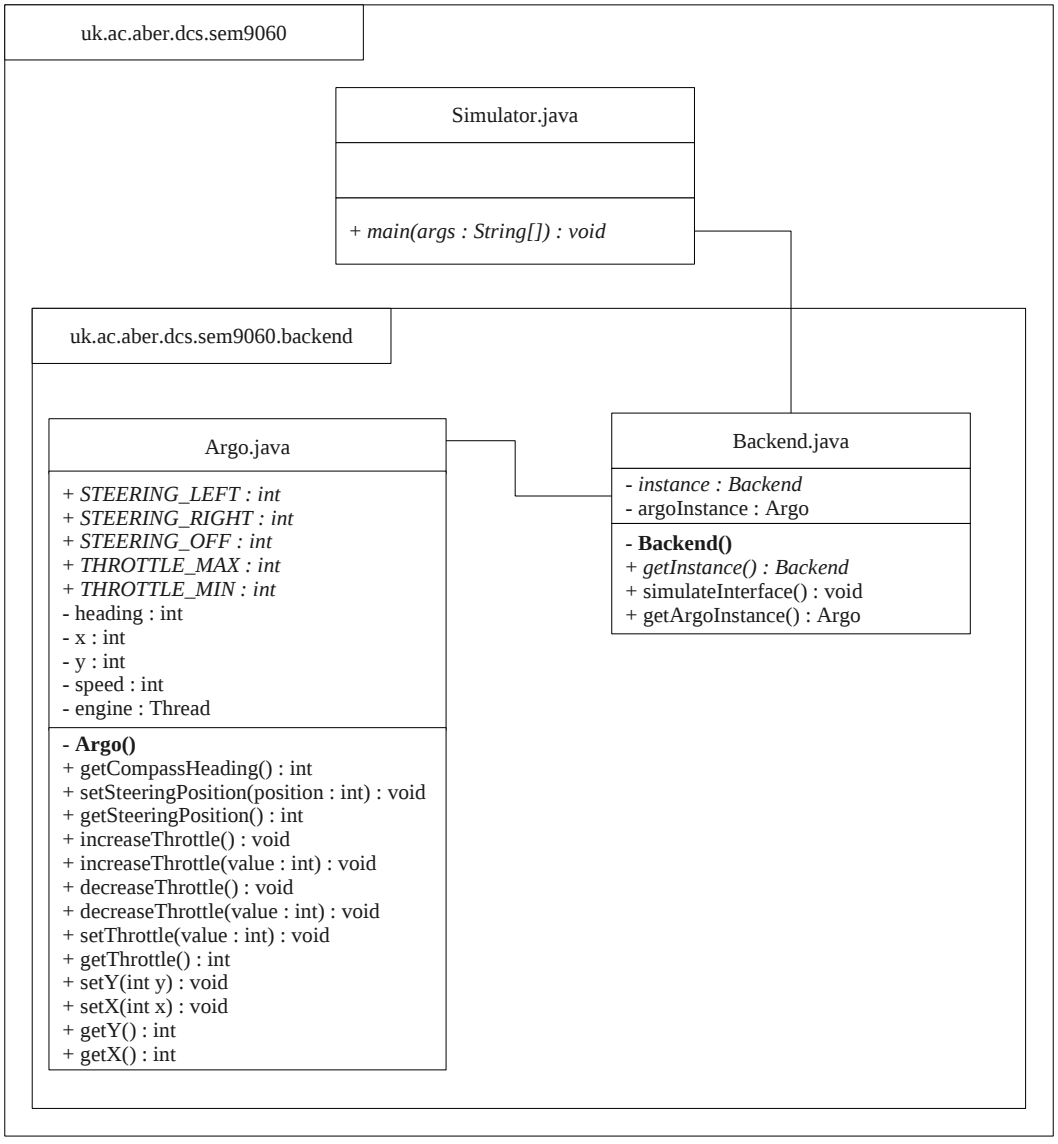


Figure 8.8: Simulator Software UML Design

8.3 Implementation

The actual coding during this sprint was again relatively simple. Surprisingly, no problems were found during the development of the physical control system on the Gumstix. The most complicated parts of the design were related to the engine thread within the simulator.

It was clear that whilst simple operations such as moving forward at zero degrees, or left at 90 degrees from North were trivial to implement, requiring only the incrementation or the decrementing of the `y` and `x` values. However, when moving at any arbitrary angle it was clear that trigonometry would be required in order to identify the appropriate new values for `x` and `y`.

The eventual algorithm used is shown below in pseudo code.

```
if (speed < THROTTLE_MIN) {
    current_speed = 1; /* Change to surface speed eventually. */

xChange = sin(current_angle) * current_speed;
yChange = cos(current_angle) * current_speed;

if (current_angle == 0) {
    y -= current_speed;
} else if (current_angle == 90) {
    x += current_speed;
} else if (current_angle == 180) {
    y += current_speed;
} else if (current_angle == 270) {
    x -= current_speed;
} else if (current_angle > 270) {
    y += y_change;
    x -= x_change;
} else if (current_angle > 180) {
    y -= x_change;
    x -= y_change;
} else if (current_angle > 90) {
    y += x_change;
    x += y_change;
} else {
    y -= yChange;
    x += xChange;
}
```

Apart from this small complication, all development work during this sprint went as per the original design.

8.4 Testing

The first step in testing was to ensure that there was no regression as a result of the new features implemented. The results of this testing is shown in the table (Figure 8.9).

Test	Feature	Expected	Result	Pass/Fail
P1	Read accelerometer.	Accelerometer values change as the hardware is Manipulated.	Changing values.	Pass
P2	Read gyro.	Gyro values change as the hardware is manipulated.	Changing values.	Pass
P3	Sent string from PIC is in correct format.	See design.	As per design.	Pass
G1	Read compass.	Display compass value.	265 degrees	Pass
G2	Read string from PIC is in correct format.	See design.	As per design.	Pass
G3	Move any servo.	Servo attached moves.	Servo moved.	Pass
S1	Read compass.	Value: 0 degrees	0 degrees	Pass

Figure 8.9: Sprint 2 Regression Testing Results, P: PIC, G: Gumstix, S: Simulator

Test	Feature	Expected	Result	Pass/Fail
G4	Press a.	Handlebars turn left.	Handlebars left.	Pass
G5	Press d.	Handlebars turn right.	Handlebars right.	Pass
G6	Press s.	Handlebars center.	Handlebars centered.	Pass
G7	Press w.	Throttle increases.	Throttle increased.	Pass
G8	Press s.	Throttle decreased to zero, handlebars center.	Throttle set to zero, handlebars centered.	Pass
G9	Press c.	Current heading displayed.	Compass heading displayed.	Pass
S2	Press a.	Handlebars turn left.	Handlebars left.	Pass
S3	Press d.	Handlebars turn right.	Handlebars right.	Pass
S4	Press s.	Handlebars centered.	Handlebars centered.	Pass
S5	Press w.	Throttle increases.	Throttle incr., x and y values changed correctly.	Pass

Figure 8.10: Sprint 2 Test Results, P: PIC, G: Gumstix, S: Simulator

Having been satisfied that no regression was apparent, the next stage was to perform testing on the features implemented during this sprint. By the end of the sprint the control system on the ARGO was at such a state that it was possible to perform live testing with the vehicle. All teleoperation functions were completed and as such manual control of the vehicle was expected to be possible. Various tests were conducted using the teleoperation mode for correct functionality, as shown in the results table (Figure 8.10).

8.5 Sprint Review

This sprint finished on time with all tasks completed. The test results are extremely promising showing that, at the very least, teleoperation of the vehicle is possible. This further suggests that the work towards automated control due to proceed in the next sprints is likely to be lucrative.

It was interesting that the most complex part of this sprint related to the development of the simulator, and not the physical control system.

Finally, according to the original plan the project remains on schedule.

Chapter 9

Sprint 3

"He who asks is a fool for five minutes, but he who does not ask remains a fool forever."

Chinese Proverb

9.1 Sprint Planning

To this point all previous sprints have started and finished on time leaving the project timings unchanged and the project running smoothly on schedule. Given this it meant that the third sprint started on time on the 4th January 2010 at the start of the examination period after the Christmas holidays.

To this point, no features from the product backlog have been removed, and no additional features have been added. Further, all previous sprints have been fully complete, leaving no features from previous sprints unimplemented. As such during this sprint the original feature set from the product backlog are expected to be completed, specifically this includes:

- 1.c.i Turn by angle algorithm.
- 1.d.vi Start high-level example algorithm.
- 1.e.i Turn by various angles, 45, 90, 180 etc.
- 2.d.i Loading map file.
- 2.d.ii Dynamic surface types.
- 2.e.i Display of the map.

9.2 Design Decisions

Few design changes were required for the design of the physical control system for this sprint. However, significant design work was required for the simulator.

9.2.1 Hardware Design

No hardware design changes were needed for this sprint.

9.2.2 PIC Firmware Design

No PIC firmware design changes were needed for this sprint.

9.2.3 Gumstix Software Design

The addition of a new file was required in order for feature 1.c.i to be implemented. The details of this file are given in this section.

control.c

This file has been designed to hold all the primary control algorithms to be used within the ARGO control system.

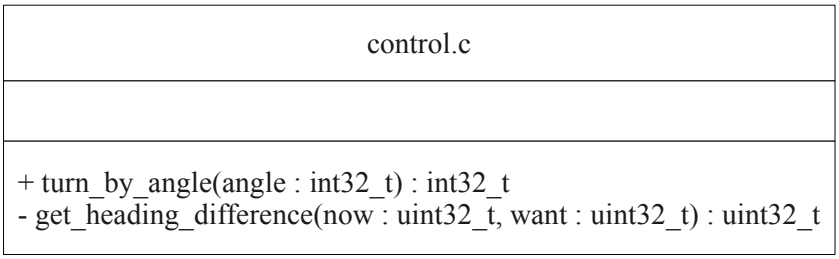


Figure 9.1: Gumstix Software Control UML Diagram

The function *turn_by_angle(angle)* will be responsible for setting the handlebar positions and the throttle control such that the vehicle turns by a specific angle relative to the start heading reading. It will use the *get_heading_difference(now, want)* function to calculate the current distance for the desired new heading.

Initially a very simple algorithm will be used to perform the turn. The new heading will be calculated according to the current heading and the desired turn angle. At this point, the control system will turn the handlebars such that the turn will never be greater than 180 degrees. Finally, the control system will continue to increase the throttle and read the compass value in a loop until the desired new heading is achieved.

Final Gumstix Software Design

The final design of the Gumstix hardware to this point is shown in Figure 9.2.

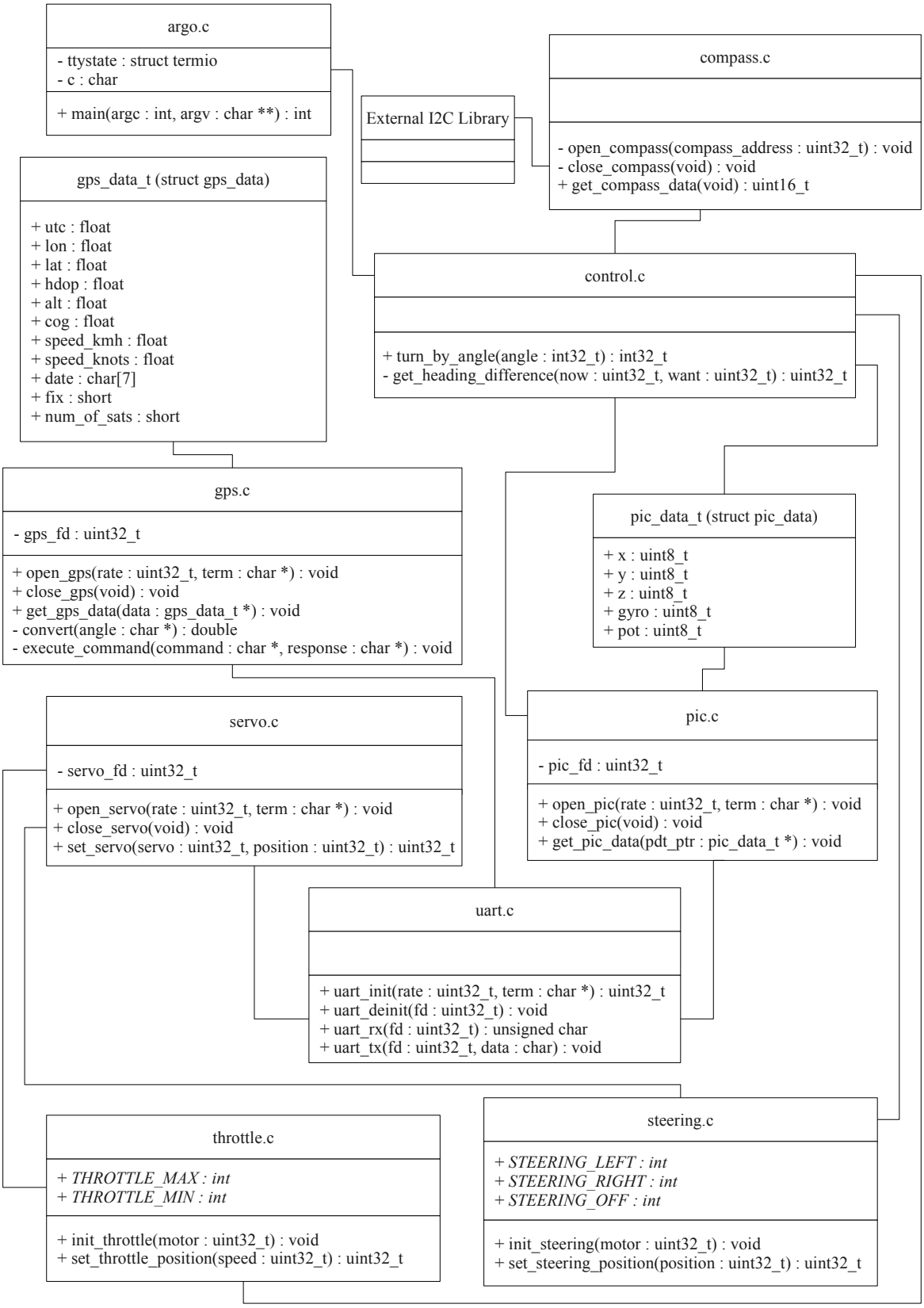


Figure 9.2: Gumstix Software UML Design

9.2.4 Simulator Software Design

Significant design work was required in order to implement the new features within the simulator. A brief overview of the design changes, and new class diagrams follows.

SurfaceType.java

The first feature to be designed was the dynamic surface types (2.d.ii). In order to achieve this it was clear that a interface class would be required that would be able to represent any possible implementation of a surface type. The resulting interface is shown in Figure 9.3.

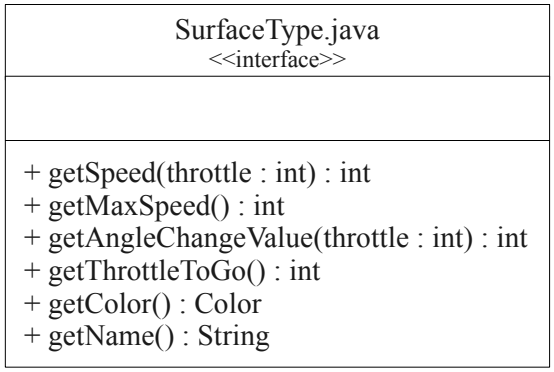


Figure 9.3: Simulator Software SurfaceType UML Diagram

The SurfaceType class has a number of methods that must be implemented in order to properly simulate different surfaces. The *getSpeed(throttle)* method is used to get the current vehicle speed given a specific throttle setting. For example, on water it is expected that a high throttle value would be needed in order to achieve any forward momentum. However, on gravel a lesser throttle value might be needed. This method will allow different surfaces to react differently, in terms of forward or reverse speed, depending on different throttle values. The actual implementation of this method is clearly therefore surface type specific. Closely related to this are the *getMaxSpeed()*, used to get the maximum possible speed the ARGO can achieve on the specific surface, and *getThrottleToGo()* which is used to get the absolute minimum that the throttle can be at before movement occurs, including turning or general driving.

The *getAngleChangeValue(throttle)* method has a similar function as the *getSpeed(throttle)* method. Given different throttle values it is likely that the speed the ARGO turns by will be different. As such, this allows for a surface type to specify what the change in heading is likely to be at given throttle values. As such, as the throttle changes during turns, the rate at which a turn takes place will fluctuate depending on the current surface.

Finally, the *getColor()* and *getName()* methods are simply for graphical display purposes. An example implementation of a Surface Type is shown in Appendix D.

Map.java

The Map class (Figure 9.4) is used to load a map file. Map files will be in the format of a XML document, and an example of a simple map file is shown in Appendix D.

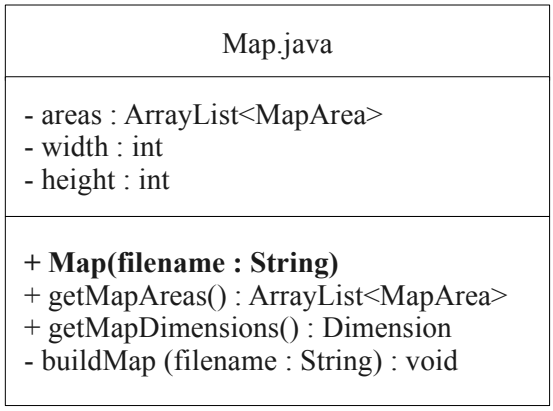


Figure 9.4: Simulator Software Map UML Diagram

Apart from having obvious attributes such as the `width` and `height` of the map the class also includes a list of all the areas within the map. These areas are discussed in further detail later in this section. However, the important method within this class is the *buildMap(filename)* method which is responsible for both reading the Map XML file and generating all the appropriate MapArea classes. These MapArea classes are then stored in the `areas` array. In order to do this the existing facilities of the Java programming language are used including the Document Object Model provided by the w3c package: *org.w3c.dom.** which is shipped with the latest versions of Java by default.

MapArea.java

The MapArea class (Figure 9.5) is used to represent the different areas of a map that are defined in any given map file. The class holds various attributes such as the `size` and `coordinates` of the area within the map, but of most important is the `surfaceType` attribute. This is to be used at a later date by the simulator to get the appropriate SurfaceType class for a given area of the map. As such, the ARGO will then be able to adapt its performance accordingly.

Backend.java

Very minor changes have been made to the Backend class such that it now loads a map file when created. This has included the addition of three methods. The first *loadMap(filename)* and *getMap()* are simple and will be used to create a new Map class and also to provide access to the class. In addition, the *getInstance(filename)* method has been added such that the first access to the backend can pass details of the location of the map file to use. Future calls to the method should ignore the file name, or more preferable is the use of the *getInstance()* function in future calls. All changes are visible in Figure 9.6.

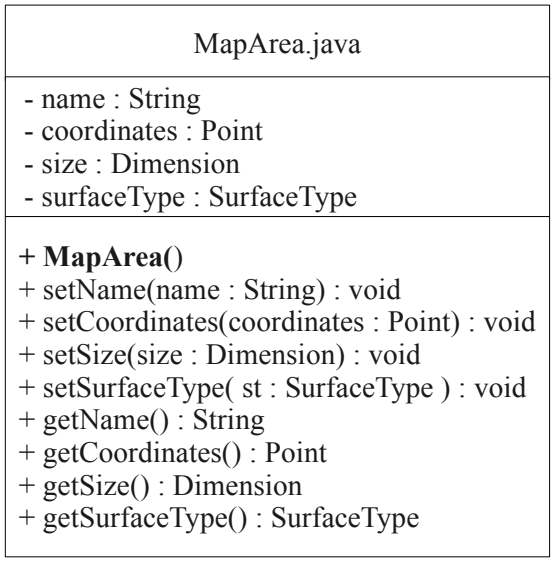


Figure 9.5: Simulator Software MapArea UML Diagram

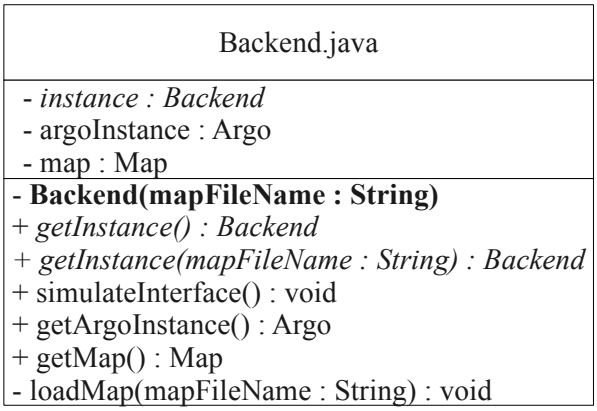


Figure 9.6: Simulator Software Backend UML Diagram

GraphicsInterface.java

The final features for this class involve the display of the map in a graphical format. The GraphicsInterface class (Figure 9.7) will hold the main `JFrame` component for the graphical user interface. As with the backend it has been decided that at any one point only one single user interface should be loaded. As such, this class is also a singleton, having a private constructor and a *getInstance()* method.

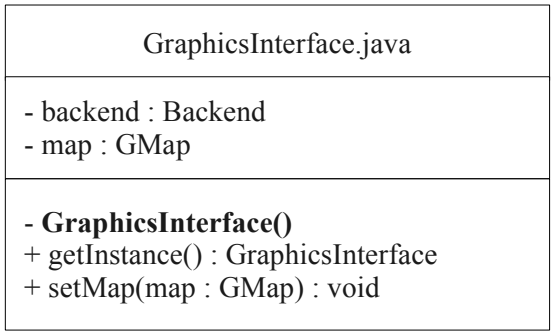


Figure 9.7: Simulator Software GraphicsInterface UML Diagram

The *setMap(map)* method is simply used to update the map `JPanel` used when displaying the graphical interface.

GMap.java

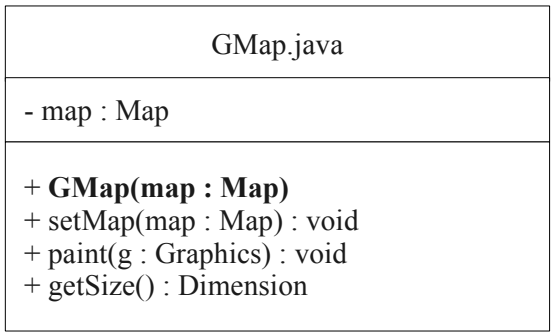


Figure 9.8: Simulator Software GMap UML Diagram

The final class required for this sprint was the `GMap` class (Figure 9.8). It is solely responsible for painting a graphical representation of the `Map` class passed into the constructor.

Final Simulator Software Design

The final design of the simulator can be seen in Figure 9.9.

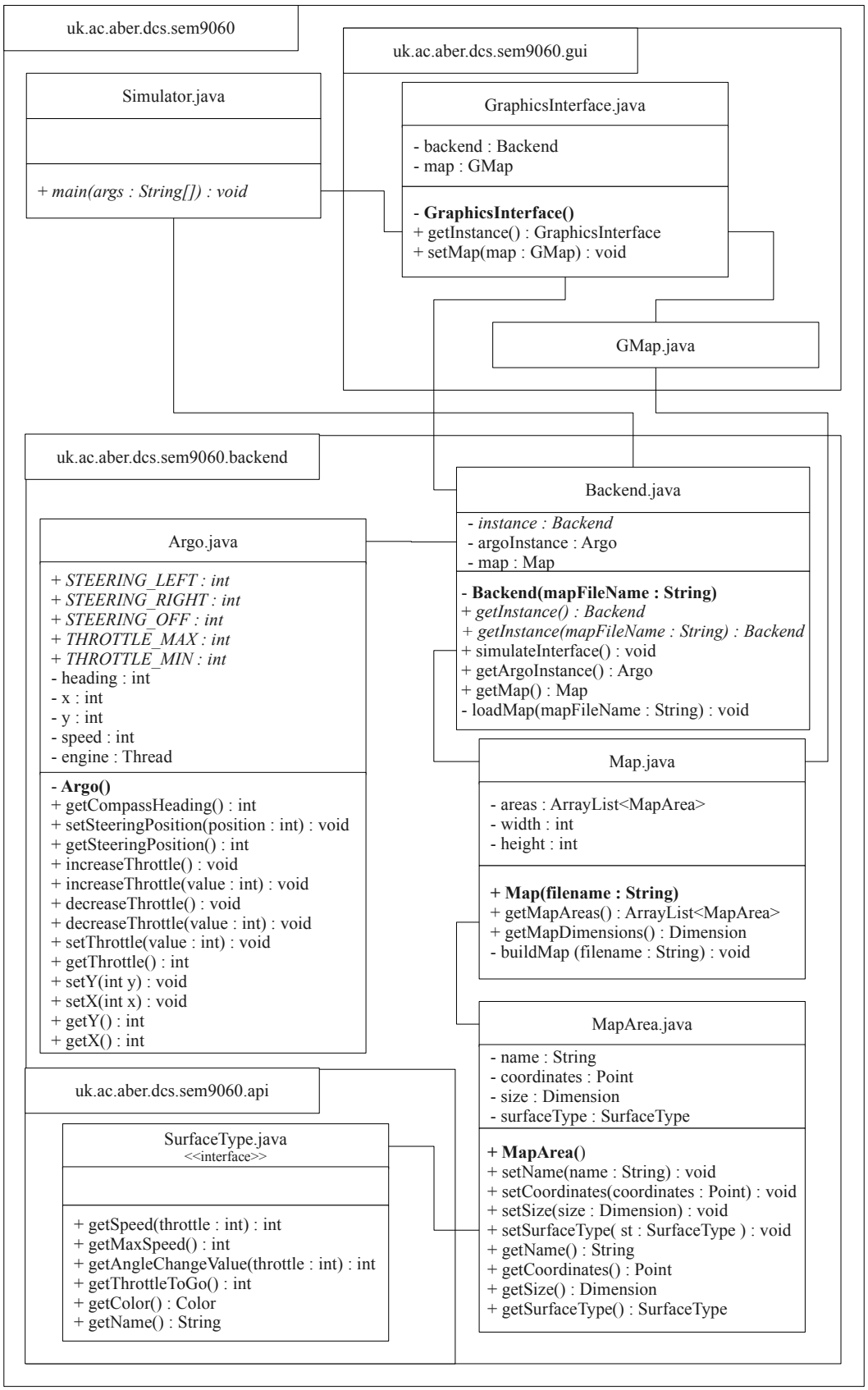


Figure 9.9: Simulator Software UML Design

9.3 Implementation

The development of the physical control system proved to be significantly more challenging during this sprint and as a result caused the sprint to go significantly over the alloted time of 2 weeks. In particular, two major problems were identified.

9.3.1 Compass Vibrations

During the development it became clear very quickly that whilst the compass had been working perfectly fine in previous sprints with the engine off, as soon as the engine was started there was a massive amount of noise. The result of this was changes in heading between 10 - 20 degrees whilst the vehicle remained stationary.

A number of different methods to try and identify the cause of the noise were attempted, including the shielding of park plugs, wrapping I2C cables in tin foil and finally using tilt-compensated compasses. The last attempt, using a tilt-compensated compass, resulted in even more noise (approximately 20 - 40 degrees of error when stationary) and it was this that made it clear that the issue was vehicle vibrations, and not electrical interference.

Various attempts were made to reduce the vibrations, including suspending the compass with elastic bands as shown in the photograph.

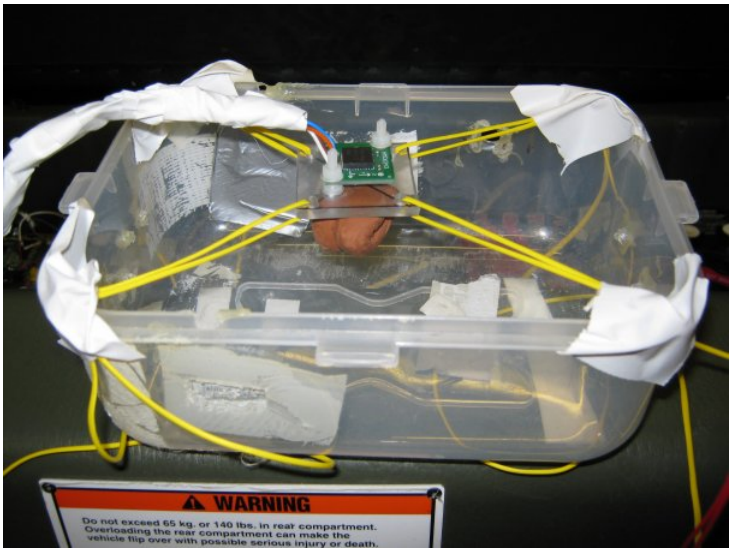


Figure 9.10: Attempts to Fix the Noisy Compass

Unfortunately a significant amount of time was wasted attempting to dampen the vibrations to the compass. Eventually it was decided that without the use of an expensive marine compass there was little that could be done. In one final attempt an average reading of five compass readings is being used to attempt to filter the compass readings somewhat. This helped to some degree, but there is still approximately 10 degrees of noise from the compass when the ARGO is standing still.

9.3.2 Rate of Turn

The second issue that caused significant annoyance during the iteration was found during turns. The rate at which the ARGO turned was considerably more than desirable and as such it was often the case that the ARGO would enter a turn and never exit, spinning on the spot for minutes at a time.

The reason for this was due to the rate of turn. As the ARGO approached the desired heading it had no method of slowing down the rate of turn. As such it would have massive overshoot past the desired heading.

A simple attempt to reduce the effects of this was used whereby the throttle was continually pulsed up and down, rather than set at a maximum value and left until the desired heading was reached. This certainly fixed some of the issues, and the ARGO behaved with more control. However, it became clear that some method of measuring the wheel rotations would be needed in order control the rate of turn more accurately. As a result of this, two additional features were added to the sprint backlog for the next sprint in an attempt to control the rate of turn. These features are discussed in Chapter 10.

9.3.3 Simulator

The simulator development work also took longer than expected due to the amount of work in the sprint. However, by the end of the sprint the simulator was starting to take shape and was capable of loading map files and displaying them in a graphical interface.

9.4 Testing

Tests were performed daily during the development stages of the physical control system, in particular during the attempts to control the rate of turn and compass noise. However no formal testing was conducted due to the nature of the functionality, the fact the sprint was already over the alloted time and due to the issues regarding the lack of controlled turns which clearly was resulting in the failure of all tests relating to turning by specific angles.

A similar situation existed for the simulator. Due to the nature of the simulator at this stage all that could be tested was that the user interface displayed correctly showing the appropriate map file. Evidence of this working is shown in Figure 9.11.

9.5 Sprint Review

Despite many issues during this sprint significant headway was made towards the final project goals. Further, two additional features were added to the product backlog in an attempt to resolve the problems discussed.

However, this sprint ran significantly over time , finishing on the 29th January (two weeks after originally anticipated), leaving little time for testing. Fortunately informal testing was conducted on a daily basis during the development. The project is now behind in terms of the original schedule. However, due to the nature of an agile methodology, and the fact that over a month was available in the initial plan for final testing and documentation work, this is not expected to have a impact in terms of the success of the project.

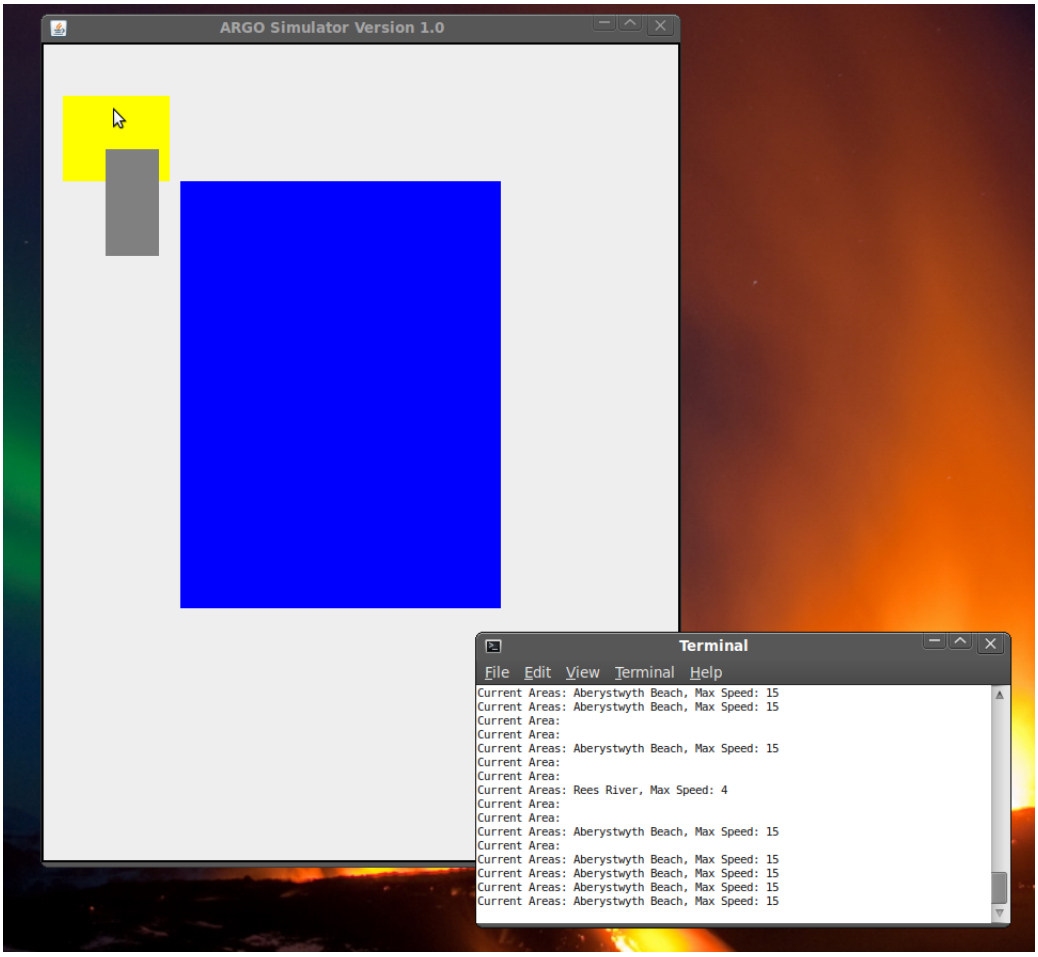


Figure 9.11: Simulator Screen Shot

Chapter 10

Sprint 4

"The intelligence of the planet is constant, and the population is growing."

Arthur C. Clarke

10.1 Sprint Planning

Unfortunately, due to the time taken during the third sprint, this sprint started two weeks later than planned on the 1st February 2010. Further, due to changes to the product backlog the features that were to be implemented during this sprint had changed.

All the original features that were to be implemented during this iteration remained the same:

- 1.c.ii Turn to angle algorithm.
- 1.e.ii Turn to various angles, 90, 180, 270, 360 etc.
- 2.b.i Turn by angle algorithm.
- 2.b.ii Turn to angle algorithm.
- 2.c.vi Start high-level example algorithms.

However two extra features were added to the product backlog for this sprint:

- 1.a.v Calculate the rate of wheel turns.
- 1.c.iii Control the rate of wheel turns.

10.2 Design Decisions

Minimal design changes are required in order to implement the product backlog during this sprint. Those design modifications and enhancements that were necessary are discussed in this section.

10.2.1 Hardware Design

During the previous sprint it became clear that it would be vital to have the ability to identify the speed at which the wheels on the ARGO rotate during different maneuvers, particularly during turn operations.

Unfortunately the current hardware design of the ARGO does not have any method of identifying the rate of wheel turns other than a speedometer. This speedometer does not provide sufficient precision for this project and further fails to identify the speed of individual sets of wheels. The information from the speedometer relates to all six wheels, not simply those on the left or right. As such a simple mechanism was needed, and has been identified, that allows for the relatively accurate measurement of left and right wheel rotations independently to a much higher degree of accuracy than the speedometer. In effect two proximity sensors will be installed, one on each chain connecting the left and right wheels to the engine. These sensors will be able to count the number of chain links that pass over a predefined period of time. Each chain link is called a 'wheel tick' from this point forward, and it has been calculated that there are a total of 7 wheel ticks per full wheel rotation.

Both sensors will feed directly into the PIC on two I/O lines. The PIC will then be responsible for counting the number of wheel ticks, and therefore wheel rotations, and sending that information to the Gumstix so that the control system can adjust the throttle appropriately during turns.

10.2.2 PIC Firmware Design

A number of possibilities existed with regards to the best method of calculating wheel rotations from the use of interrupts to a statically timed control loop. Numerous of these different approaches were attempted during the implementation process in order to identify the most appropriate, as discussed later in the implementation section. However, it was eventually decided that the best solution was to use a statically timed loop that, rather than counting the actual number of wheel ticks, calculated the amount of time that had passed since the previous wheel tick. This then allowed the frequency of wheel rotations to be calculated.

Unfortunately the PIC only has support for up to 16-bits, and as such all timers and integers have a maximum range between 0 and 64,999 when unsigned. This proved to be insufficient, in terms of the range of values, to count all wheel ticks. As such, two 16-bit integers were used for each wheel where the second integer represented an overflow of the first 16-bit number. As the first number reached 64,999 it was reset and the second number was incremented by one. The result of the correct calculations with these two 16-bit numbers is a 32-bit integer with sufficient range to count all wheel ticks.

As a result of the changes to the PIC, modifications were once again needed to the data string to be sent back to the Gumstix from the PIC. The format of the new data string in shown in Figure 10.1.

10.2.3 Gumstix Software Design

pic.c and pic_data_t

Having the information regarding the time since the last wheel rotation meant that minor modifications were required to the PIC code. First, two additions were made to the

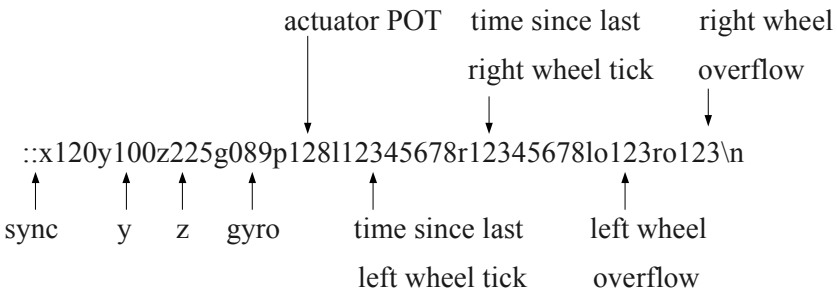


Figure 10.1: Revised PIC Data String

pic_data_t structure to hold the new information. The two attributes `left_wheels` and `right_wheels` in the diagram (Figure 10.2) hold the left and right wheel rotations per second.

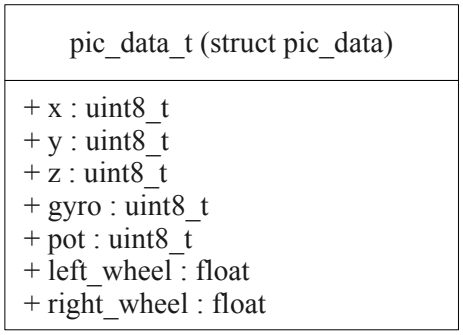


Figure 10.2: Gumstix Software pic_data_t UML Diagram

The rotations are calculated using the formula shown (Eq. 10.1) where r is the number of rotations per second, t is the number of timer ticks since the last wheel rotation as provided by the value `l` or `r` in the PIC data string. v represents the number of times the value t has overflowed and comes from `lo` and `ro` in the PIC data string. k_1 is a constant set at 64999 and is the last possible value that t can be before it overflows and is reset to zero. Finally, k_2 is a second constant (0.0002) and represents the precise number of seconds per timer tick.

$$r = \frac{1}{(t + vk_1) \cdot k_2} \tag{10.1}$$

control.c

Only a single design change was made to the control.c file, and that was the inclusion of the *turn_to_angle(angle)* function. The design of this function was made as simple as possible, simply calling the *turn_by_angle(angle)* function with the appropriate heading difference between the current heading of the ARGO and the target heading.

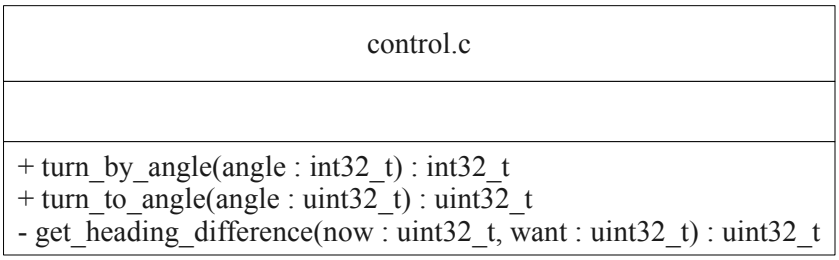


Figure 10.3: Gumstix Software control.c UML Diagram

However, significant work was undertaken to the *turn_by_angle(angle)* function to improve accuracy. Details of this are given in the implementation section, however the final design uses a PI controller to achieve controlled turns.

Final Gumstix Software Design

A final overview of the Gumstix software design is shown in Figure 10.5.

10.2.4 Simulator Software Design

Control.java

The design work for the simulator in this sprint is minimal and simply follows the existing design in place in the physical control system resulting in the addition of a Control class (Figure 10.4).

Like the control.c file in the physical ARGO control system, three methods exist: *getHeadingDifference(now, want)* is used to get the heading difference between two headings, *turnByAngle(angle)* is used to turn by a given angle relative to the current heading and finally *turnToAngle(angle)* is used to turn to a specific heading.

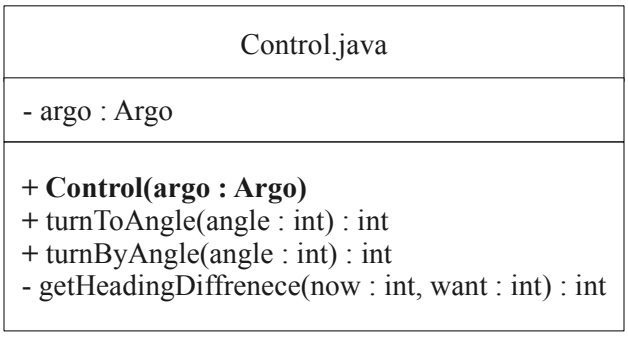


Figure 10.4: Simulator Software Control UML Diagram

Final PIC Software Design

The final design of the PIC software to this point is shown in Figure 10.6.

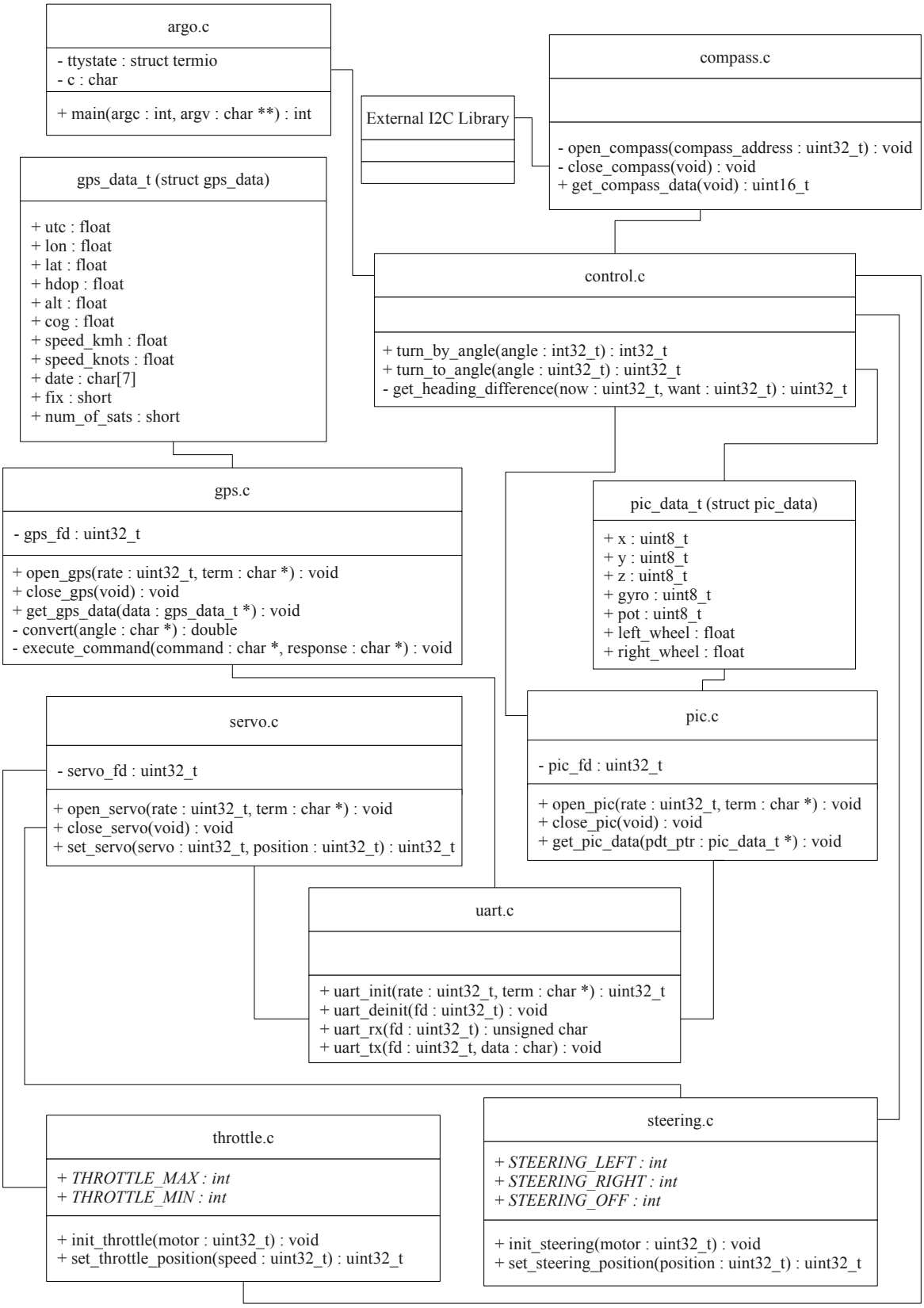


Figure 10.5: Gumstix Software UML Design

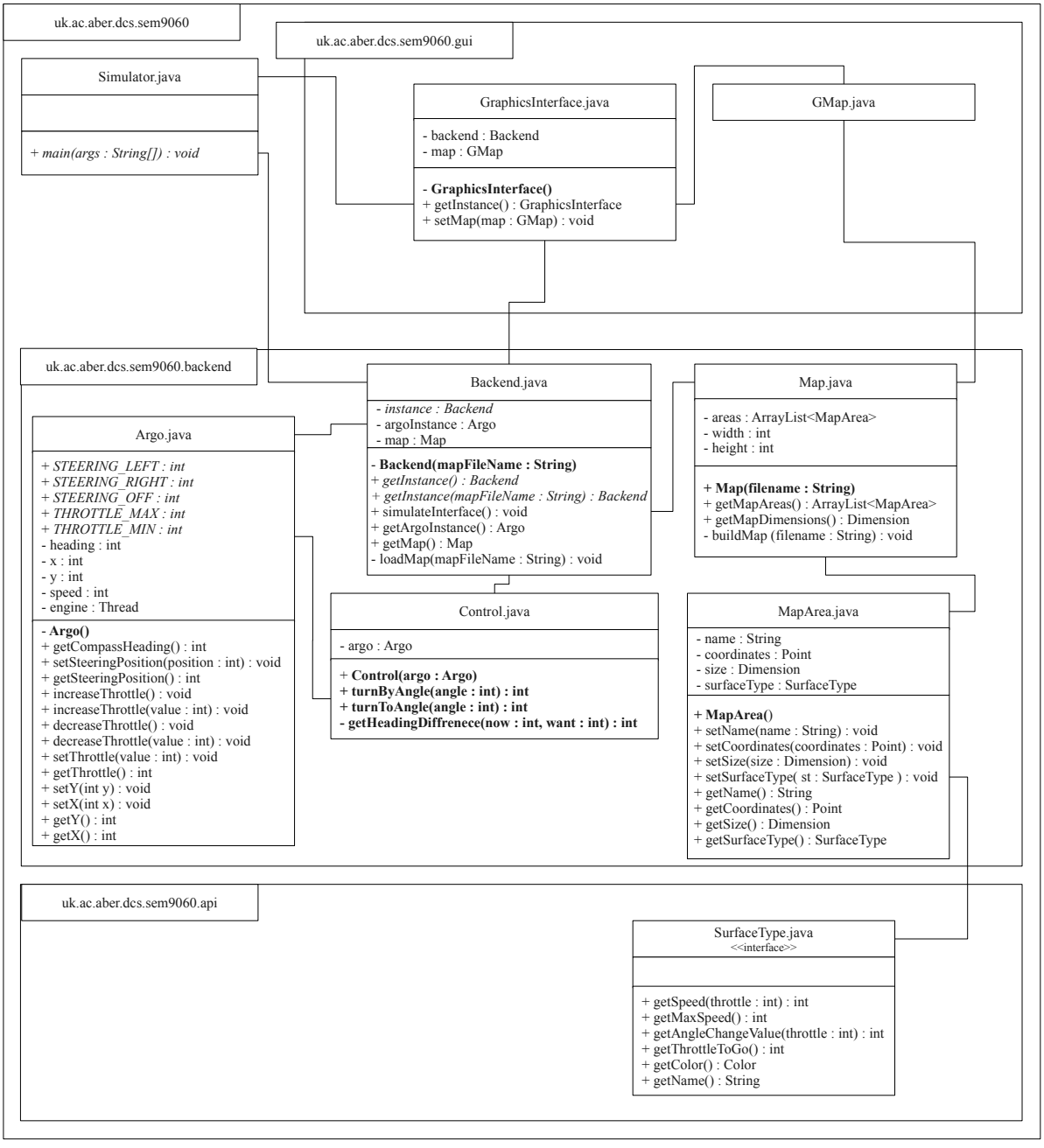


Figure 10.6: Simulator Software UML Design

10.3 Implementation

10.3.1 Wheel Rotation Counting

It was initially thought that an interrupt based approach would be most appropriate for counting wheel ticks. As a change is identified by the sensors an interrupt would be generated incrementing a counter. However, this proved to be unacceptable due to the fact that it did not provide guaranteed execution time and, as a result of this, it was often the case that the PIC was so busy dealing with interrupts that it was unable to spend time transmitting data over the serial line.

As such it became clear that a fixed time loop system would be required. This meant that the PIC software needed to keep a record of the previous state of the sensors on both the left and right wheels (i.e. high or low), and then compare the current reading against the previous. Any changes would then result in a counter being incremented. A timer would then be used to generate an interrupt once a second to reset the counters to zero. As such every second it would be clear how many wheel rotations there had been. However, this again was found to be unacceptable for two reasons. First, it means that if the wheels stop turning it can take a whole second before this is noticed and second the interrupt generated every second often occurred during the transmission of serial data causing significant issues regarding which value should be sent over the serial line (the previous value or the new value of zero) and also with regards to corrupted data arriving at the Gumstix.

A final solution was identified without the use of any interrupts. Instead of counting the number of wheel ticks per second, the time since the last wheel tick was counted and the frequency was then calculated as detailed in the design section (Eq. 10.1). This proved to be a much better solution, capable of counting all wheel rotations in a fixed timed loop algorithm, without having any impact on the transmission of data to the Gumstix. The final algorithm used is shown below.

```
1  while(1) {
2
3          timer = ReadTimer0();
4          left_wheel += timer;
5          right_wheel += timer;
6
7          if (right_wheel > 64999) {
8                  right_wheel = 0;
9                  rwover++;
10         }
11
12         if (left_wheel > 64999) {
13                 left_wheel = 0;
14                 lwover++;
15         }
16
17         if (rwover > rwoverprint)
18                 rwoverprint = rwover;
19
20         if (lwover > lwoverprint)
21                 lwoverprint = lwover;
22
23         if (left_wheel > lwprint)
24                 lwprint = left_wheel;
```

```
25
26          if (right_wheel > rwprint)
27                  rwprint = right_wheel;
28
29          if (PORTBbits.RB6 != left_last) {
30                  left_last = PORTBbits.RB6;
31                  lwprint = left_wheel;
32                  lwoverprint = lwover;
33                  left_wheel = 0;
34                  lwover = 0;
35          }
36
37          if (PORTBbits.RB7 != right_last) {
38                  right_last = PORTBbits.RB7;
39                  rwprint = right_wheel;
40                  rwoverprint = rwover;
41                  right_wheel = 0;
42                  rwover = 0;
43          }
44          WriteTimer0(0);
45  }
```

10.3.2 Proportional Control During Turning

After the problems during the previous sprint and the clear need for a reduced turn rate a number of attempts were made to adjust the rate of turns.

Firstly, the fluctuating throttle 'solution' from the previous sprint was modified such that the throttle would be increased until the wheels actually began to rotate, rather than to an arbitrary maximum figure. At this point, the throttle was instantly reduced to zero again. Unfortunately, whilst providing a minor improvement in control over the previous solution, it was still not sufficient to get fully controlled turns. At this point it became clear that what was really needed was a method of controlling the throttle proportionally to the target heading.

As such a simple P and later a PI controller was implemented. The proportional aspect came from the distance to the target heading. This value then results in a target wheel rotation speed that is related to the heading distance.

Using this target wheel rotation speed the current error in terms of the current wheel rotations verses desired wheel rotations is calculated, and the result is used to produce a throttle adjustment value. This value is then added to (or deducted from) the current throttle, resulting in the integral component of the PI controller. As such, the further the distance from the heading, the greater the error in wheel rotations and so the greater the error in the throttle setting, thus the larger the resulting adjustments of throttle value used to compensate. Further, failure for any change in heading to occur after throttle adjustments simply result in the continual increase in the throttle until the vehicle moves.

A simple trial-and-error method was used to tune the PI controller, and two gain values were calculated. The first, 0.0075, is the gain for the proportional component, and 10.0 was the resulting gain for the integral component. The result of this tuning was the ability of the throttle to respond quickly to changes in wheel rotations, whilst the wheel rotations remained at appropriate levels for controlled turns with relation to the distance to the target headings.

Compass Calibration

After the issues relating to the compass in the previous sprint, further issues were identified. Whilst the ARGO was now performing turns with reasonable accuracy (and certainly in a controlled manner) it was apparent that a lot of the turns failed to result in the ARGO pointing in the correct heading.

Initially it was thought that this was still due to vibrations. However, it became apparent that the compass was simply uncalibrated, as can be seen in the graph (Figure 10.7).

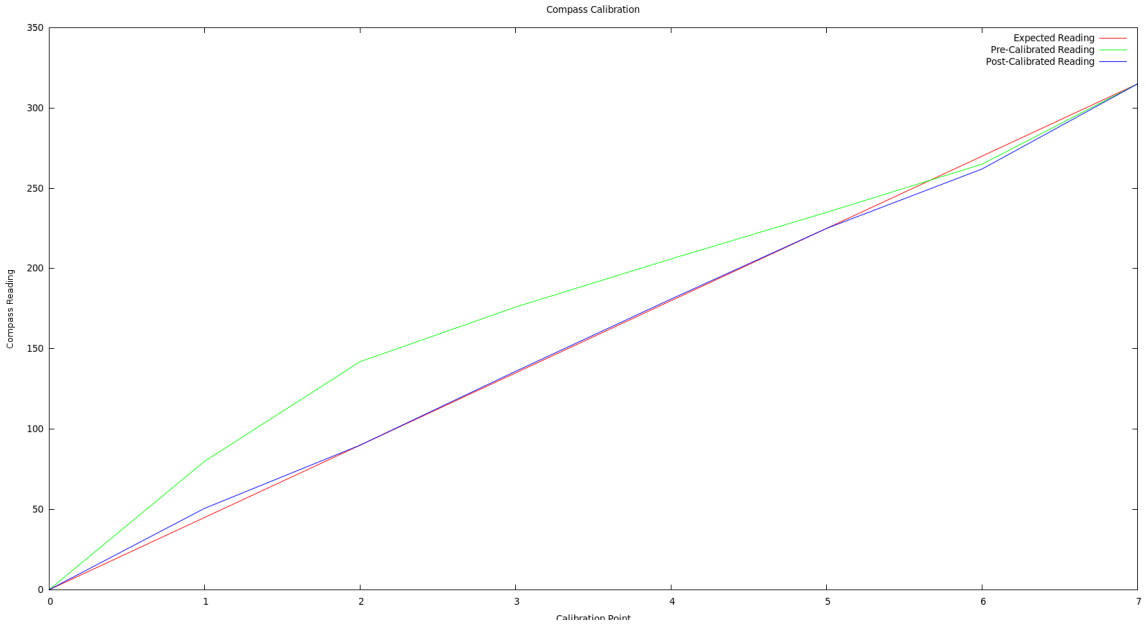


Figure 10.7: Compass Calibration

The resulting compass calibration required the use of two linear formula. The first (Eq. 10.2) is used for compass values between 40 and 112 degrees, and the second (Eq. 10.3) is used for all compass values between 112 and 275 degrees.

$$h_{new} = 0.63 \cdot h_{old} \tag{10.2}$$

$$h_{new} = 225 - ((235 - h_{old}) \cdot 1.52) \tag{10.3}$$

10.4 Testing

As with previous sprints testing took place on a daily basis during the development of the software and in particular during the development of the physical control system. All previous tests were re-run in order to ensure that no features implemented during this sprint had caused regression, and the results are shown in the table (Figure 10.8).

A brief overview of additional tests required for this sprint, and the results, is provided in the table (Figure 10.9).

Test	Feature	Expected	Result	Pass/Fail
P1	Read accelerometer.	Accelerometer values change as the hardware is Manipulated.	Changing values.	Pass
P2	Read gyro.	Gyro values change as the hardware is manipulated.	Changing values.	Pass
P3	Sent string from PIC is in correct format.	See design.	As per design.	Pass
G1	Read compass.	Display compass value.	265 degrees	Pass
G2	Read string from PIC is in correct format.	See design.	As per design.	Pass
G3	Move any servo.	Servo attached moves.	Servo moved.	Pass
G4	Press a.	Handlebars turn left.	Handlebars left.	Pass
G5	Press d.	Handlebars turn right.	Handlebars right.	Pass
G6	Press s.	Handlebars center.	Handlebars centered.	Pass
G7	Press w.	Throttle increases.	Throttle increased.	Pass
G8	Press s.	Throttle decreased to zero, handlebars center.	Throttle set to zero, handlebars centered.	Pass
G9	Press c.	Current heading displayed.	Compass heading displayed.	Pass
S1	Read compass.	Value: 0 degrees	0 degrees	Pass
S2	Press a.	Handlebars turn left.	Handlebars left.	Pass
S3	Press d.	Handlebars turn right.	Handlebars right.	Pass
S4	Press s.	Handlebars centered.	Handlebars centered.	Pass
S5	Press w.	Throttle increases.	Throttle incr., x and y values changed correctly.	Pass

Figure 10.8: Sprint 4 Regression Testing Results, P: PIC, G: Gumstix, S: Simulator

Test	Feature	Expected	Result	Pass/Fail
P4	Wheel rotations counting, moving left wheel.	Left wheel rotations have appropriate changing values.	Changing values.	Pass
P5	Wheel rotations counting, moving right wheel.	Right wheel rotations have appropriate changing values.	Changing values.	Pass
G10	Press g.	Various angles achieved.	Various angles achieved.	Pass
S6	Press g.	Various angles achieved.	Various angles achieved.	Pass

Figure 10.9: Sprint 4 Test Results, P: PIC, G: Gumstix, S: Simulator

10.5 Sprint Review

Despite starting two weeks late and having more work than initially planned this sprint has proven to be significantly more successful than the previous sprint. In order to include testing, the sprint went over the estimated time by one week, making the sprint three weeks long and finishing on the 20th February 2010.

During the sprint a number of important decisions were made with regards to achieving acceptable levels of control of the ARGO resulting in the completion of all functional requirements for the physical ARGO control system and proving that control of such a vehicle is indeed possible to a reasonable level even without expensive components once sufficient software calibration has taken place.

By the end of the sprint all original features, and additional features, had been completed and were shown to be in good order via testing. Further, no regression had resulted from the addition of the new features. However, the project was now three weeks behind the original schedule.

Chapter 11

Sprint 5

"If you focus on results, you will never change. If you focus on change, you will get results."

Jack Dixon

11.1 Sprint Planning

The fifth and final sprint of the project started on the 22nd February 2010, three weeks later than originally planned. However, the original requirements of the project for this sprint were trivial - simply involving the enhancement of the graphical components of the simulator environment:

- 2.e.ii Continually updating display of the ARGO and its current position on the map.
- 2.e.iii Display of sensor data and the current status of the ARGO, such as the coordinates, heading, speed and throttle etc.

Despite being late according to the original project plan sufficient time existed for the addition of a secondary objective into the product backlog. It was decided that the implementation of a high-level simplistic GPS navigation algorithm using simple way-point navigation would be easy to develop thanks to the now proven and functional low level control software. The lead to the addition of one final feature, 1.e.iii:

- 1.e.iii Develop simple GPS way-point navigation techniques.

Whilst there has been an additional feature added to the product backlog, the original primary objectives are of most importance. As such, whilst this chapter deals with the physical implementation details, and therefore the secondary objective, first in order to remain consistent with all the previous chapters, during the development stage the simulator aspects took first priority during this sprint.

11.2 Design Decisions

11.2.1 Hardware Design

No hardware design changes were needed for this sprint.

11.2.2 PIC Firmware Design

No PIC firmware changes were needed for this sprint.

11.2.3 Gumstix Software Design

A number of small changes were required to the design of the Gumstix software for this sprint. Specifically, the introduction of GPS navigation code into the control.c class and minor modifications to the gps.c file.

gps.c

In order to achieve GPS navigation two pieces of information were necessary. First, it was important to be able to get the appropriate turn angle given the current heading of the ARGO so that the vehicle could turn to face the new GPS point. This was achieved with the new function *gps_get_heading(curr, dest)*. The mathematics for this calculation are shown in Equation 11.1.

$$\theta = atan2(sin(\Delta long) \cdot cos(lat2), cos(lat_1 \cdot sin(lat_2) - sin(lat_1) \cdot cos(lat_2) \cdot cos(\Delta long))^1 \tag{11.1}$$

The other information vital is the current distance to the point. Clearly, if the distance is 0 meters the position has been reached and the vehicle should not re-attempt to navigate to the position. The function *gps_get_distnace(curr, dest)* deals with the calculation of distance between two GPS positions via Equation 11.2 and 11.3.

$$a = sin^2(\frac{\Delta lat}{2}) + cos(lat_1) \cdot cos(lat_2) \cdot sin^2(\frac{\Delta lon}{2}) \tag{11.2}$$

$$d = (R \cdot (2 \cdot atan2(\sqrt{a}, \sqrt{1-a})) \cdot k^1 \tag{11.3}$$

In Equation 11.3, R is the approximate radius of the Earth (6371km) and k is a constant of 1000 used in order to change the units from kilometers to meters. The resulting UML diagram representing the new gps.c file is shown in Figure 11.1.

control.c

The addition of one function was made to the control.c file: *navigate_to_gps_point(lon, lat)*. It is used to perform the actual navigation from one GPS point to another using all other control functions previously developed including the *turn_to_angle(angle)* and the new GPS functions.

argo.c

Whilst no design changes were made to this file, two additions occurred. First, a simple loop will be implemented at the start of the main method to read in a simple GPS data file (see Appendix D for an example) which holds way points and a menu item 'n' for navigate will also be added to start autonomous navigation to the GPS coordinates specified in the data file.

[1]Both formula from: `http://www.movable-type.co.uk/scripts/latlong.html`, Last accessed: 22nd April 2010.

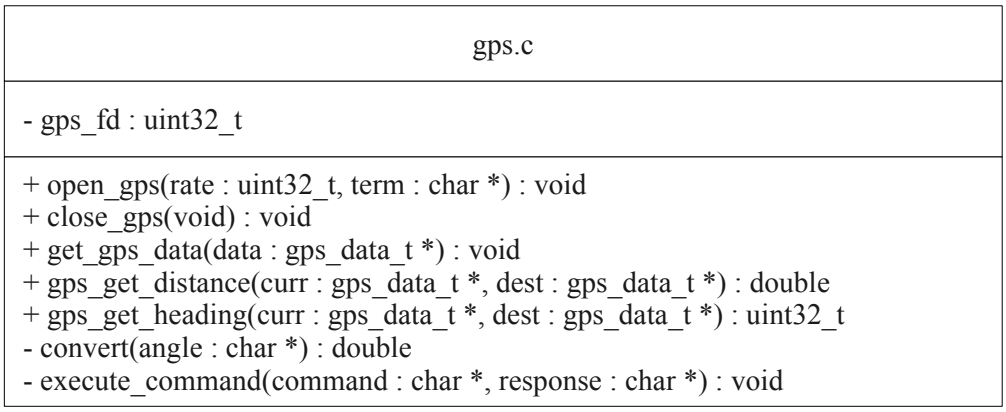


Figure 11.1: Gumstix Software GPS UML Diagram.

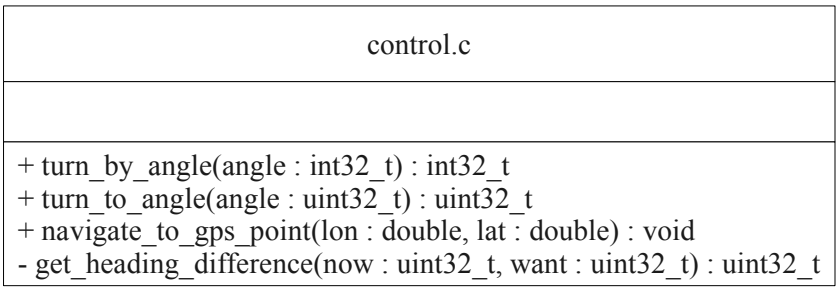


Figure 11.2: Gumstix Software Control UML Diagram.

Final Gumstix Software Design

The final software design for the Gumstix is shown in Figure 11.3

11.2.4 Simulator Software Design

During this sprint two new features were to be added, both of which solely affect the graphical user interface of the simulator. Given that all the work relating to moving the ARGO takes place in the backend, the work during this sprint was anticipated to be trivial. The final design work for this sprint is shown detailed in the remainder of this chapter.

DataPanel.java

The DataPanel class (Figure 11.4) has the simple job of displaying data regarding the current ARGO operations. This data includes:

- Current Speed
- Current x and y Coordinates
- Current Heading
- Surface Type
- Maximum Possible Speed
- Current Throttle Position
- Handlebar Position

The *updateDisplay()* method is used to update this information with the latest figures from the backend. This happens once every 1 millisecond via a call from the main `GraphicsInterface` class.

GArgo.java

The GArgo class (Figure 11.5) is simply used to draw the current position of the ARGO, and its orientation, on the map. The *draw(g)* method is called by the `GMap` class whenever the map is re-drawn.

Final Simulator Software Design

Figure 11.6 shows a UML diagram of the final simulator design.

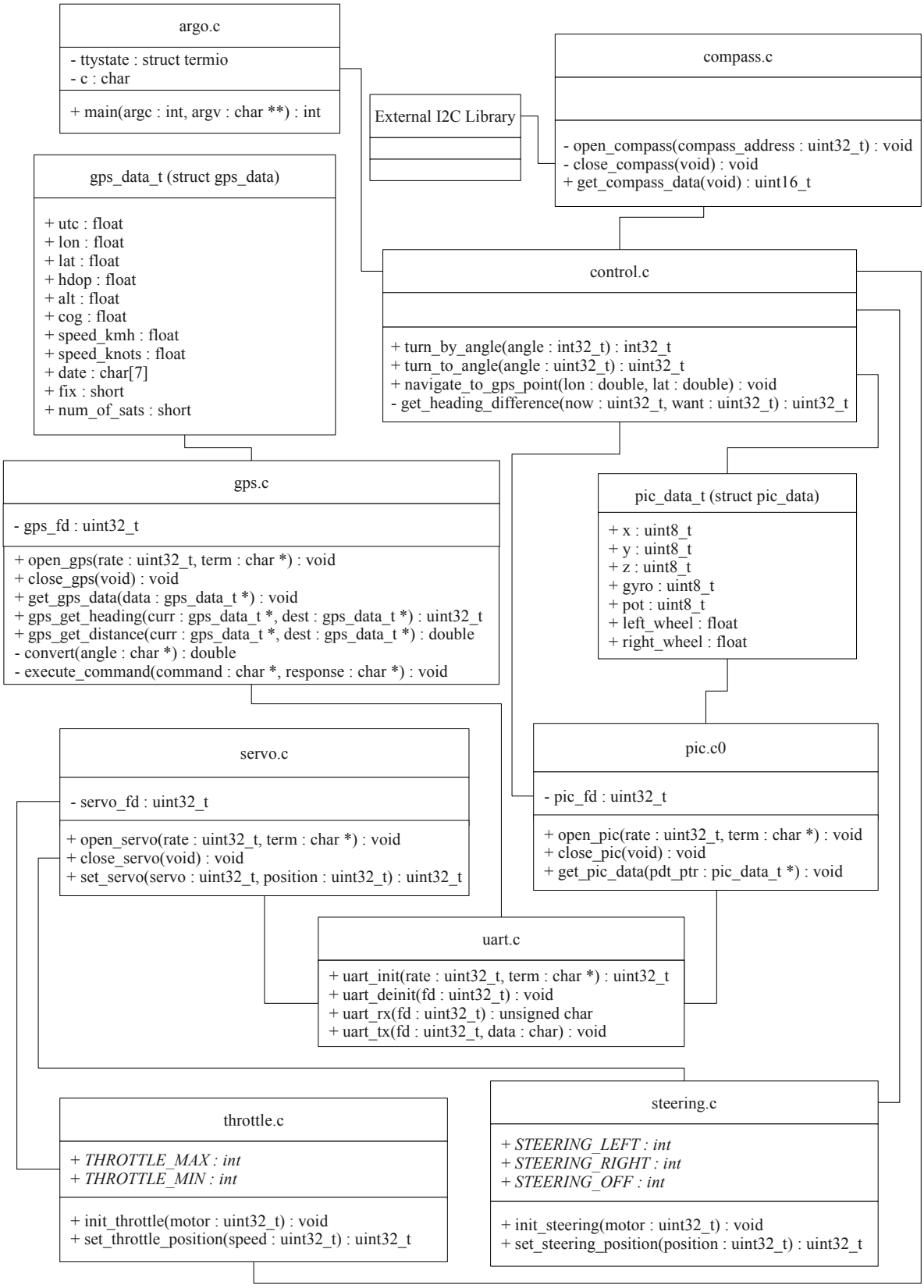


Figure 11.3: Gumstix Software UML Design

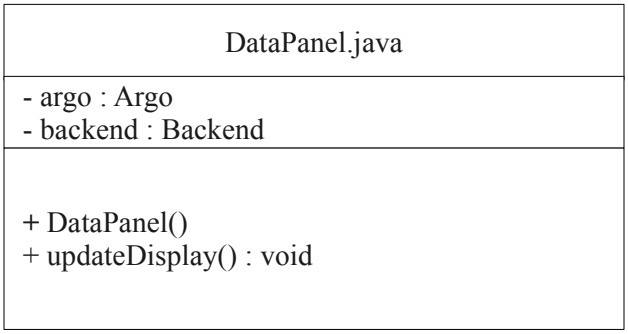


Figure 11.4: Simulator Software DataPanel UML Diagram

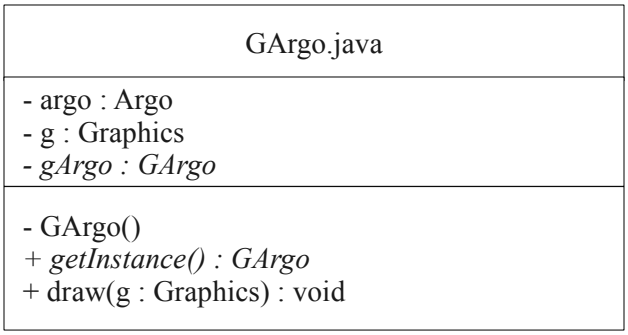


Figure 11.5: Simulator Software GArgo UML Diagram

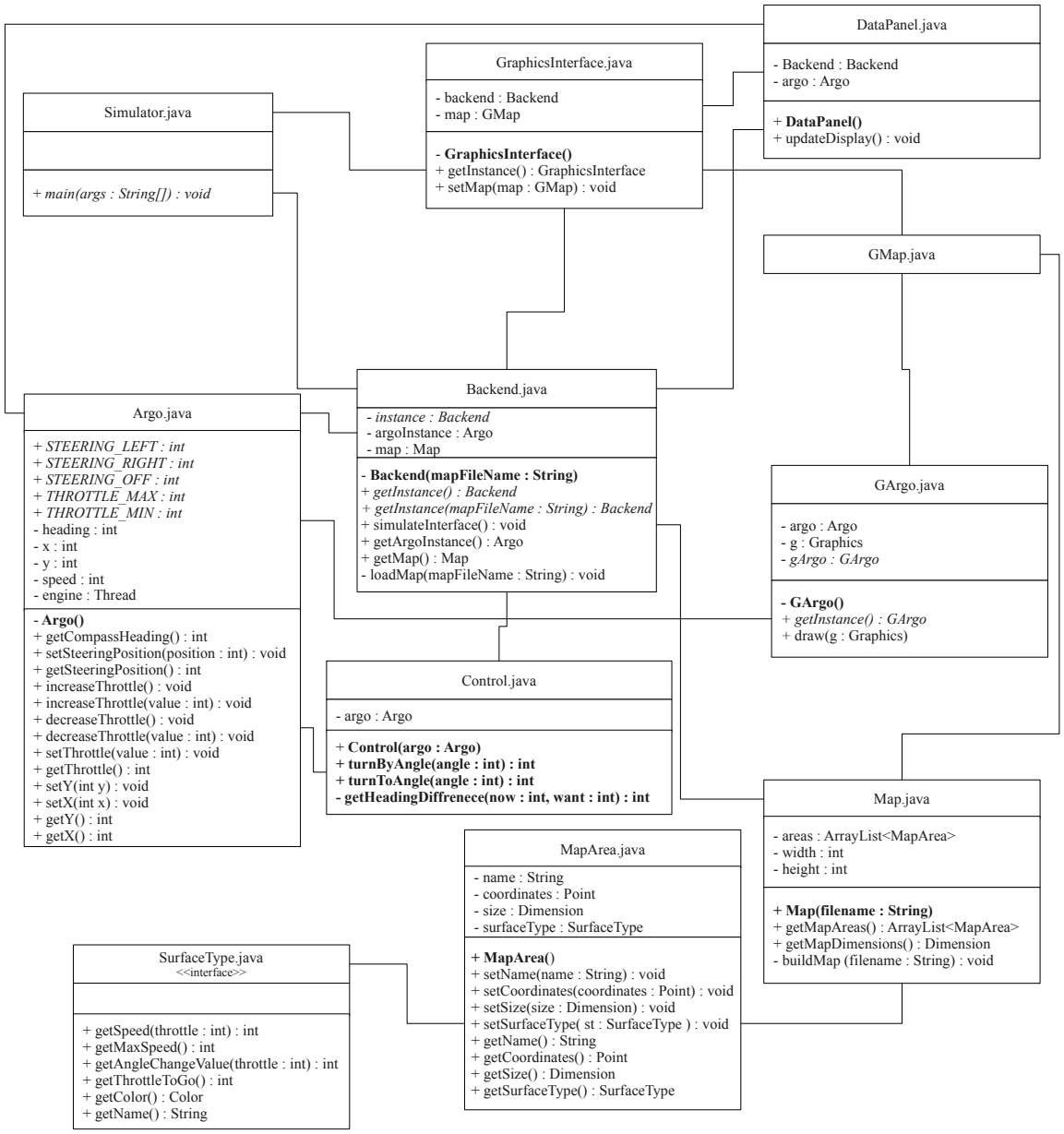


Figure 11.6: Simulator Software UML Design

11.3 Implementation

11.3.1 GPS Navigation

Despite the anticipated simplicity the GPS navigation actually proved to be more difficult than originally expected. The primary cause for problems was the continued noise in the compass. As the vehicle moved the compass vibrations would often cause the control algorithm to 'think' that the ARGO was off course - often by a factor of 10 - 20 degrees. This would cause the ARGO to continually turn on the spot, first left, then right etc in an attempt to get precisely on heading. As such the vehicle made very little progress forwards towards the final destination.

Fortunately, a primitive solution was found to this problem. The control algorithm was updated such that the ARGO would only attempt to correct course in two circumstances:

1. The ARGO was travelling in completely the wrong direction, and the distance to the final way-point was increasing rather than decreasing.

2. The ARGO had moved forwards, in any direction, at least 5 meters since the last course correction.

This seemed to fix the problems seen and the ARGO now navigates to GPS points with few course corrections. The resulting control algorithm used is shown below.

```
1  void navigate_to_gps_point(double lon, double lat) {
2
3          gps_data_t current, new;
4          uint32_t compass, heading;
5          float distanceStart, distanceCurrent;
6          extern uint32_t DIS_TO_WAYPOINT, ANG_TO_WAYPOINT;
7
8          get_gps_data(&current);
9          new.lon = lon;
10         new.lat = lat;
11
12         distanceStart = gps_get_distance(&current, &new);
13         distanceCurrent = gps_get_distance(&current, &new);
14
15         while (distanceCurrent > DIS_TO_WAYPOINT) {
16
17                 get_gps_data(&current);
18                 compass = get_compass_data();
19                 heading = gps_get_heading(&current, &new);
20
21                 distanceCurrent = gps_get_distance(&current, &new);
22
23                 while((get_heading_difference(compass, heading)
24                   > ANG_TO_WAYPOINT && (distanceStart - distanceCurrent > 5.0))
25                         || (distanceCurrent > distanceStart)) {
26
27                         distanceStart = gps_get_distance(&current, &new);
28                         set_throttle_position(THROTTLE_MIN);
29                         turn_to_angle(heading);
30                         set_steering_position(STEERING_OFF);
```

```
31
32                          compass = get_compass_data();
33
34                  }
35
36                  set_throttle_position(73);
37
38          }
39
40          set_throttle_position(THROTTLE_MIN);
41          set_steering_position(STEERING_OFF);
42
43  }
```

Finally, testing and tuning took place before it was decided that a reasonable distance to waypoint (`DIS_TO_WAYPOINT`) figure was 7 meters. Further, it was decied that the use of 15 degrees for the angle to waypoint (`ANG_TO_WAYPOINT`) parameter was suitable. Both these seemed to work best in terms of the accuracy and repeatability when navigating to the same GPS position.

11.3.2 Simulator

The simulator development work was as expected and proved to be exceedingly trivial, taking approximately two hours to complete! No problems were identified during the development work worth noting.

Figure 11.7 shows the final output of the simulator whilst performing turns.

11.4 Testing

As with all previous sprints, regression testing was conducted to ensure that no new features had caused faults to occur. The results are shown in Figure 11.4.

Having confirmed that no features were regressed during this sprint testing took place to ensure that all the new features were correctly functioning. Given the nature of these features, no automation was possible, and so all tests are 'visual' in nature and the results are shown in Figure 11.4.

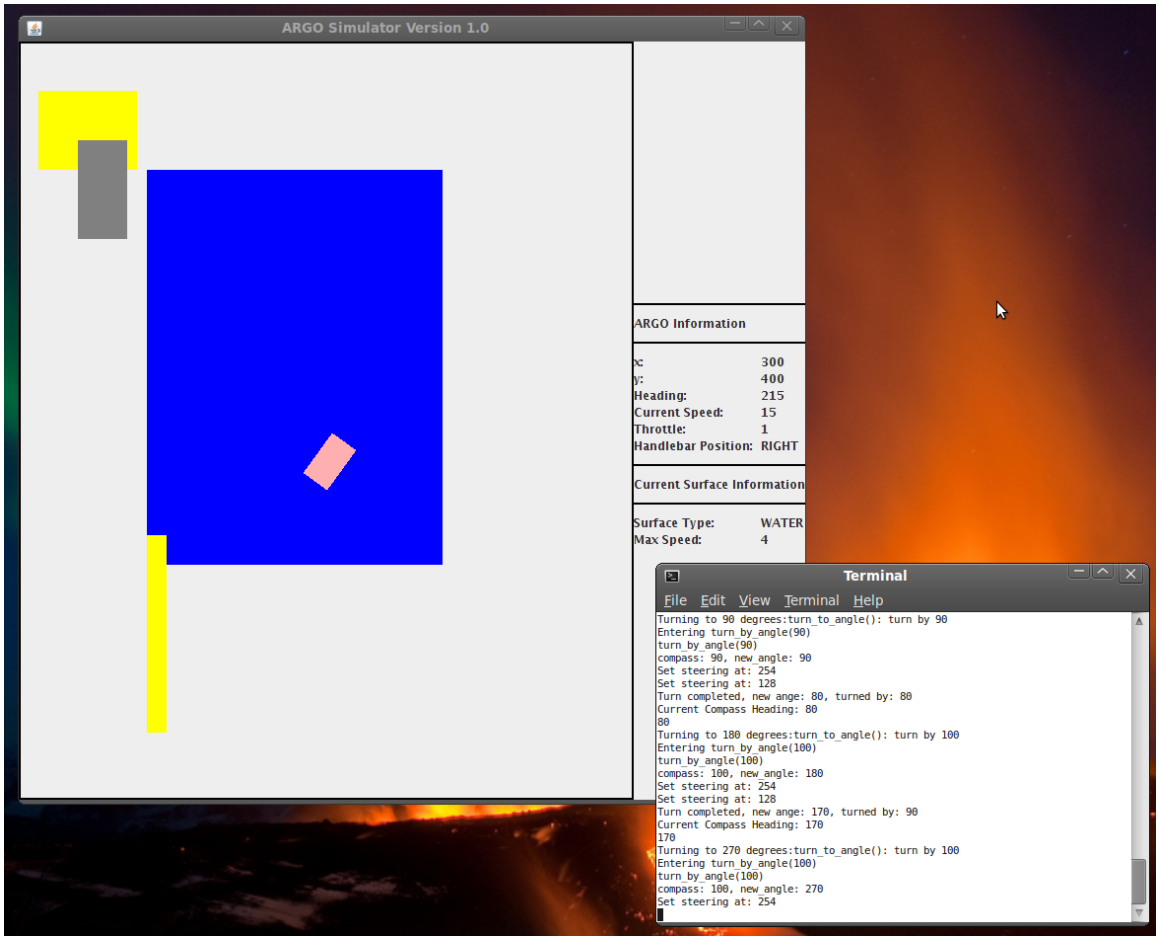


Figure 11.7: Simulator Screen Shot Performing Turns

Test	Feature	Expected	Result	Pass/Fail
P1	Read accelerometer.	Accelerometer values change as the hardware is Manipulated.	Changing values.	Pass
P2	Read gyro.	Gyro values change as the hardware is manipulated.	Changing values.	Pass
P3	Sent string from PIC is in correct format.	See design.	As per design.	Pass
P4	Wheel rotations counting, moving left wheel.	Left wheel rotations have appropriate changing values.	Changing values.	Pass
P5	Wheel rotations counting, moving right wheel.	Right wheel rotations have appropriate changing values.	Changing values.	Pass
G1	Read compass.	Display compass value.	265 degrees	Pass
G2	Read string from PIC is in correct format.	See design.	As per design.	Pass
G3	Move any servo.	Servo attached moves.	Servo moved.	Pass
G4	Press a.	Handlebars turn left.	Handlebars left.	Pass
G5	Press d.	Handlebars turn right.	Handlebars right.	Pass
G6	Press s.	Handlebars center.	Handlebars centered.	Pass
G7	Press w.	Throttle increases.	Throttle increased.	Pass
G8	Press s.	Throttle decreased to zero, handlebars center.	Throttle set to zero, handlebars centered.	Pass
G9	Press c.	Current heading displayed.	Compass heading displayed.	Pass

Test	Feature	Expected	Result	Pass/Fail
G10	Press g.	Various angles achieved.	Various angles achieved.	Pass
S1	Read compass.	Value: 0 degrees	0 degrees	Pass
S2	Press a.	Handlebars turn left.	Handlebars left.	Pass
S3	Press d.	Handlebars turn right.	Handlebars right.	Pass
S4	Press s.	Handlebars centered.	Handlebars centered.	Pass
S5	Press w.	Throttle increases.	Throttle incr., x and y values changed correctly.	Pass
S6	Press g.	Various angles achieved.	Various angles achieved.	Pass

Figure 11.8: Sprint 5 Regression Testing Results, P:PIC, G: Gumstix, S: Simulator

Test	Feature	Expected	Result	Pass/Fail
G11	GPS Navigation	Navigate between three different waypoints specified.	+/- 7m distance from GPS points achieved.	Pass
S7	Argo Displays	Argo is displayed and moves in the appropriate directions.	Moving argo.	Pass
S8	Data Display	Appropriate and correct data is displayed and updated in the GUI.	Data correct.	Pass

Figure 11.9: Sprint 5: Testing Results, P: PIC, G: Gumstix, S: Simulator

11.5 Sprint Review

This final sprint represented the end of the development work for the project. Once again, despite starting late, the sprint was highly successful with all tests passing and all features complete. Further, it is pleasing to have primitive GPS navigation available at the end of this sprint as it clearly helps to show that the control and automation of a vehicle such as the ARGO is possible. The only issues during this sprint were related, once again, to the compass hardware used. However, even given these issues, the performance has clearly been sufficient for simple navigation tasks.

Whilst the simulator aspect during this sprint took very little time, the GPS code took much longer than anticipated. As such this sprint also ran over time by one week. This puts the whole project four weeks behind the original plan. Overall this sprint is considered to be a great success and the project is in such a state that data can be gathered in order to assist with the final research objectives.

Chapter 12

Data Analysis

"No amount of experimentation can ever prove me right; a single experiment can prove me wrong."

Albert Einstein

12.1 Data Gathering

The first step towards answering the research objectives of this project was to gather sufficient data. Some initial data relating to human-driven testing was already available from the investigative work completed at the beginning of this project. Further, a significant amount of data existed from the many tests completed during the development stages.

However, the data gathered during the development work was clearly not sufficient for data analysis. Whilst it may show interesting results, one set of data cannot be compared against another set of data. The conditions were often different, tuning parameters often changed between testing, details were not recorded of the precise surface conditions or of anomalies during testing which could produce perplexing results.

As such it was clear that a set of final testing would need to be conducted under much more controlled circumstances. Data gathered would have to be identical across all tests and details regarding the surface type and conditions (wet, dry, damp etc) would also be needed. Further, it was important to decide what data would actually be appropriate - would it have been sufficient to gather only 90 degree turns or would 45 degree turns have been better. Further, does the use of only 45 degree turns result in different characteristics than the use of 90 or 180 degree turns?

Eventually, after some thought, a data gathering strategy was decided. Simple paper records of all three tests are attached to this dissertation in Appendix B. The testing involved the use of 45, 90 and 180 degree turns on three separate surfaces including tarmac, grass and gravel. In addition to the testing sheets log files were collected for each test which held data regarding the wheel rotations, compass heading, throttle values, gyroscope and accelerometer readings. Due to the size of these data files they have not been attached directly to this dissertation. However, they are available on the provided CD.

12.2 Data Processing

Having acquired all the data from the individual tests each turn during the testing was processed into individual files using a simple shell script:

```
1 cat $1 | sed s/:/=/g > $1.out
2 cat $1.out | sed s/[a-z_]*=//g > $1
3 cat $1 | sed s/[a-z]*//g > $1.out
4 cat $1.out | sed s/"␣"//g > $1
5 cat $1 | sed s/","/",␣"/g > $1.out
6 mv $1.out $1
```

This left the data in the comma separated format required by tools such as GNU Plot. The format of the data at this point was:

- gyroscope,
- accelerometer x-axis,
- accelerometer y-axis,
- accelerometer z-axis,
- current throttle speed,
- current heading,
- difference between desired heading and current heading,
- left wheel rotations per second,
- right wheel rotations per second,
- target wheel rotations per second,
- difference between right/left (depending on turn direction) rotations and the target wheel rotations.

This data was then processed using GNU Plot to generate various graphs to allow for data analysis. Some of these graphs are included in Appendix C along with hand written analysis notes.

12.2.1 Gravel

Figure 12.2.1 shows turn data plotted for a 45 degree turn on gravel. Specifically it shows a turn from 45 degrees to 90 degrees from North. The first thing that is noticeable regarding this data is that there is considerable variation in the wheel rotation speed with large spikes, yet there are only very small and progressive changes in heading during the turn.

In addition to this, the compass (and as such target wheel rotations and heading difference) seems to be exceedingly noisy with constantly changing readings +/- 5 degrees either side of the actual heading.

Both these characteristics may be as a result of the surface type. On gravel it was observed during testing that the wheels would often get stuck or spin on the loose surface. This would cause the throttle to immediately decrease because the wheel rotations would be significantly over the target and eventually the throttle would increase again until the wheels started to grip and move at the correct rotational speed. At this point the vehicle would turn quite quickly until the heading was reached, or the wheels once again hit loose ground.

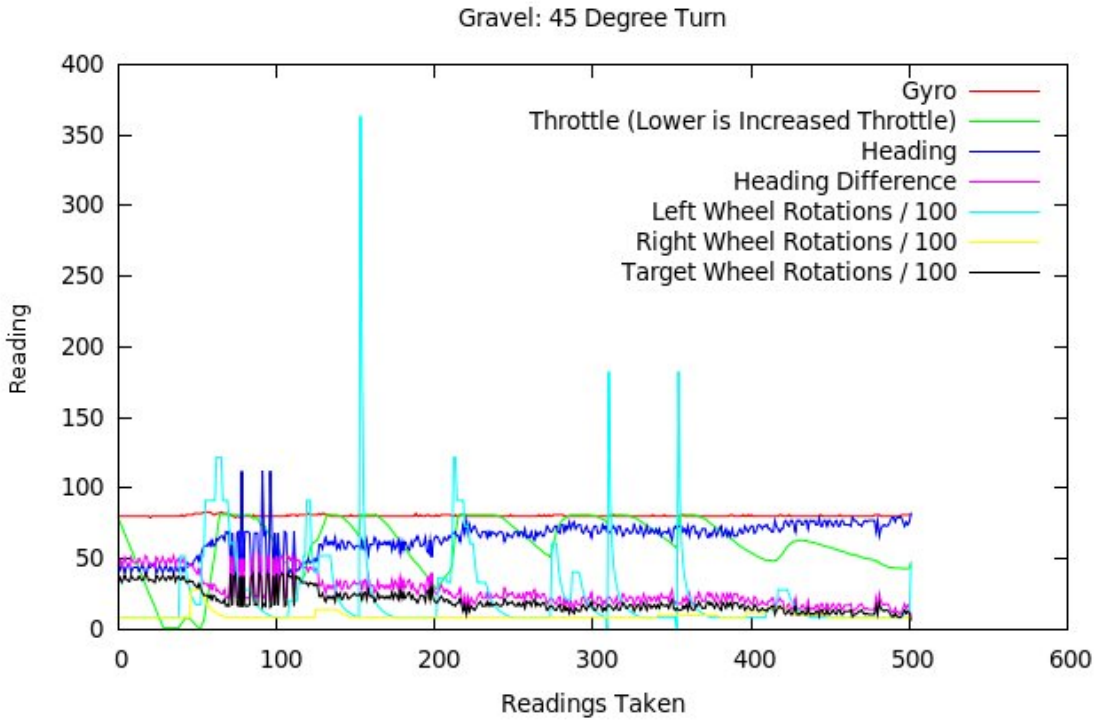


Figure 12.1: A Typical 45 Degree Turn on Gravel

This behaviour on gravel is further seen in the throttle value. On all the graphs for gravel the throttle is seen to increase and decrease with considerable uniformity. This is particularly well seen in the 180 degree turn data (Figure 12.2.1) where the throttle never reaches maximum but approximately half speed before wheel rotations shoot far past the target resulting in the instant reduction of throttle.

It is interesting that the response of the engine seems to be quite fast on gravel - at least in terms of acceleration. Very little throttle is needed before massive wheel rotations occur. However, it is also clear that whilst wheel rotations increase they remain high before the vehicle actually moves. This is particularly well seen in the 180 degree turn between 50 and 200 readings. The wheel rotations are high for a considerable duration whilst the change in heading seems to have large steps, rather than smooth changes.

It seems to this point that the data in the graphs quite closely represents the observed behaviour.

12.2.2 Grass

All the graphs for Grass (Figures 12.2.2, 12.2.2 and 12.2.2) show considerably different characteristics compared to gravel. Unlike with gravel there is a significant delay between the increase in throttle and the increase in wheel rations. Even on the smaller turns the throttle reached the maximum value and stayed there for approximately 30 readings before any wheel rotations occurred. This is significantly different to gravel where minor changes in throttle had a relatively quick effect on wheel rotations.

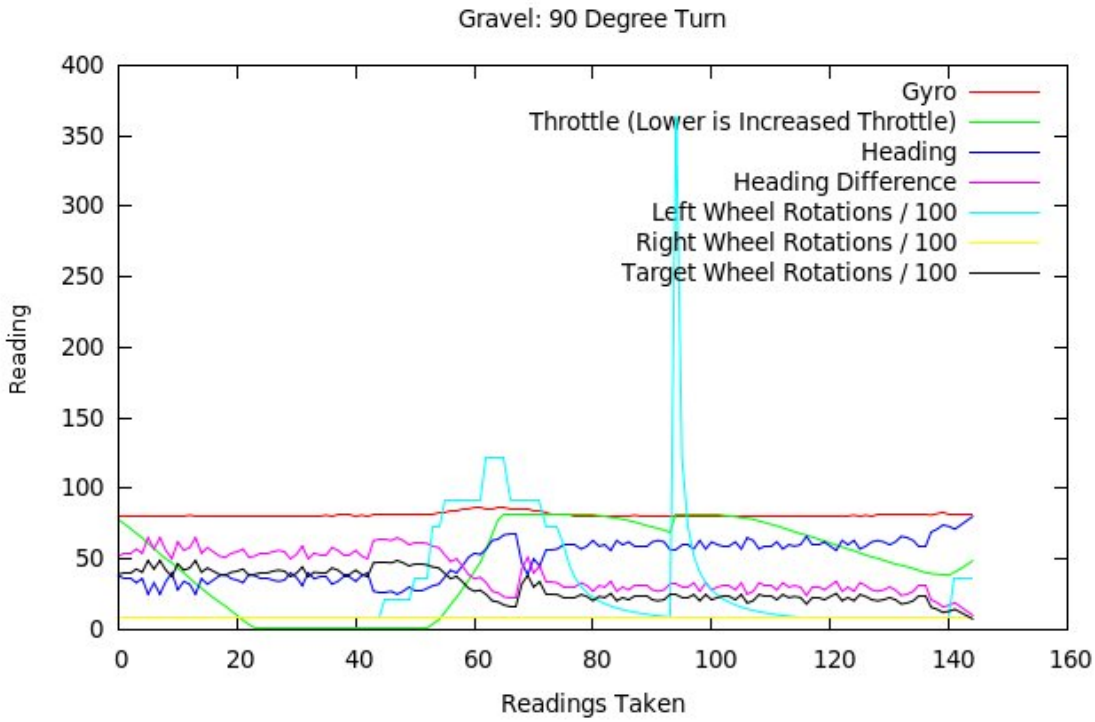


Figure 12.2: A Typical 90 Degree Turn on Gravel

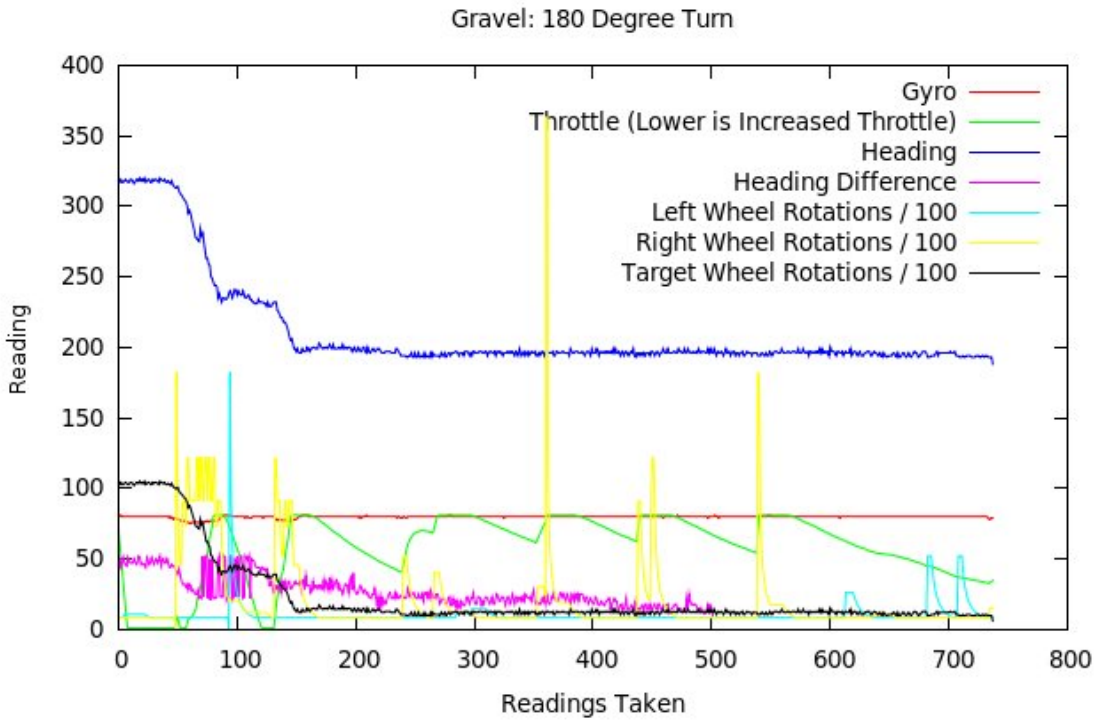


Figure 12.3: A Typical 180 Degree Turn on Gravel

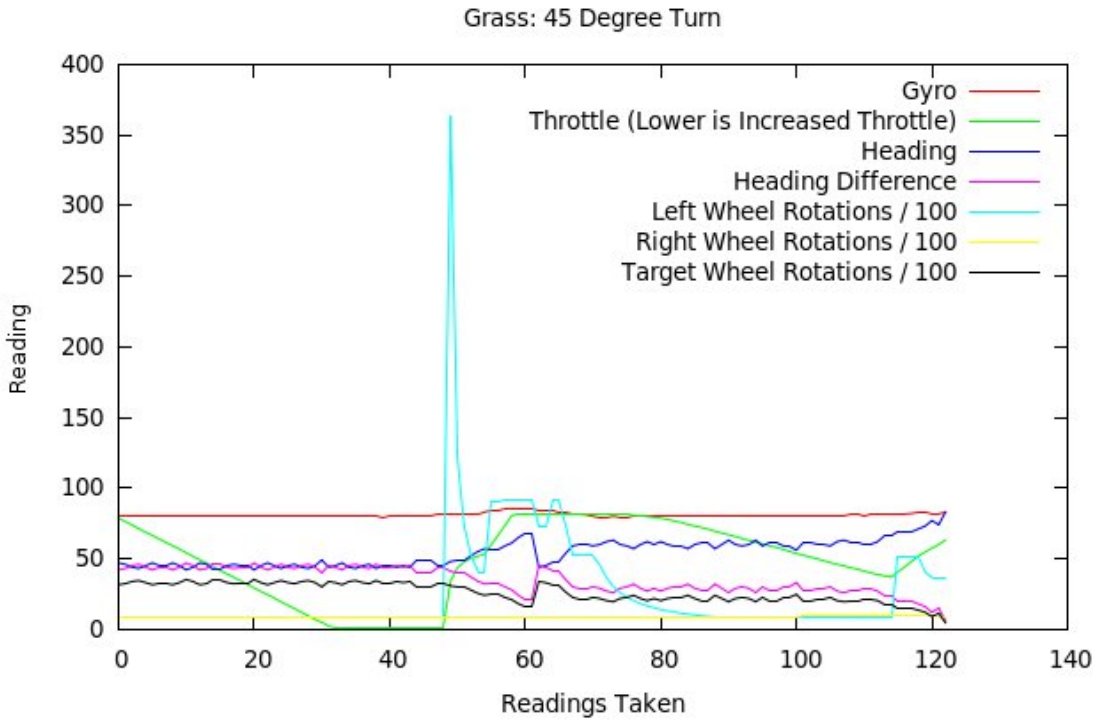


Figure 12.4: A Typical 45 Degree Turn on Grass

Further differences are evident in terms of the effects of those wheel rotations. As soon as the wheels begin to rotate the compass heading changes. This is once again different than on gravel where wheels spin on loose gravel before changes in heading.

Finally, and probably due to the relatively slow reaction of the wheels, there seems to be significantly fewer fluctuations in the throttle on grass than on gravel.

12.2.3 Tarmac

The data on tarmac is of significant interest. First, it should be noted that the surface was particularly grainy with a strange almost sand like coating of tarmac on top of the normal road surface. Further, the test data for tarmac was on a considerable slope due to the limited and constrained testing environments available. As such the data may not be directly comparable with grass or gravel.

However, most of the data gathered on the tarmac seems to have very similar characteristics to the grass surface. Once again long delays are seen between the increase in throttle and turns actually taking place. Further similarities include the rate of turns. As with grass, once the wheels spin the heading changes with incredible speed.

It is interesting to see the effects of the slope within the data. Whilst most of the turn data looks incredibly like grass some parts exhibit quite different characterises. For example during the 180 degree turn (Figure 12.2.3), readings 50 - 100 show a considerable duration where the wheel rotations are high. This is clearly different to grass, and looks more like gravel. Further, the turn rate during this is relatively constant and progressive - not quick

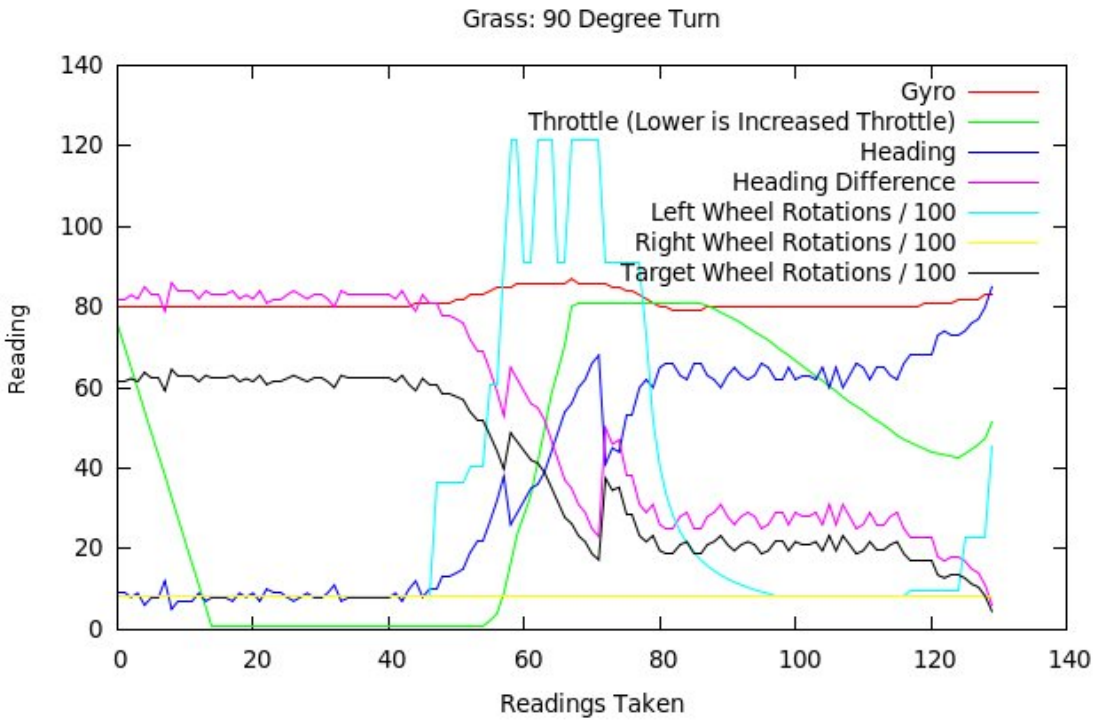


Figure 12.5: A Typical 90 Degree Turn on Grass

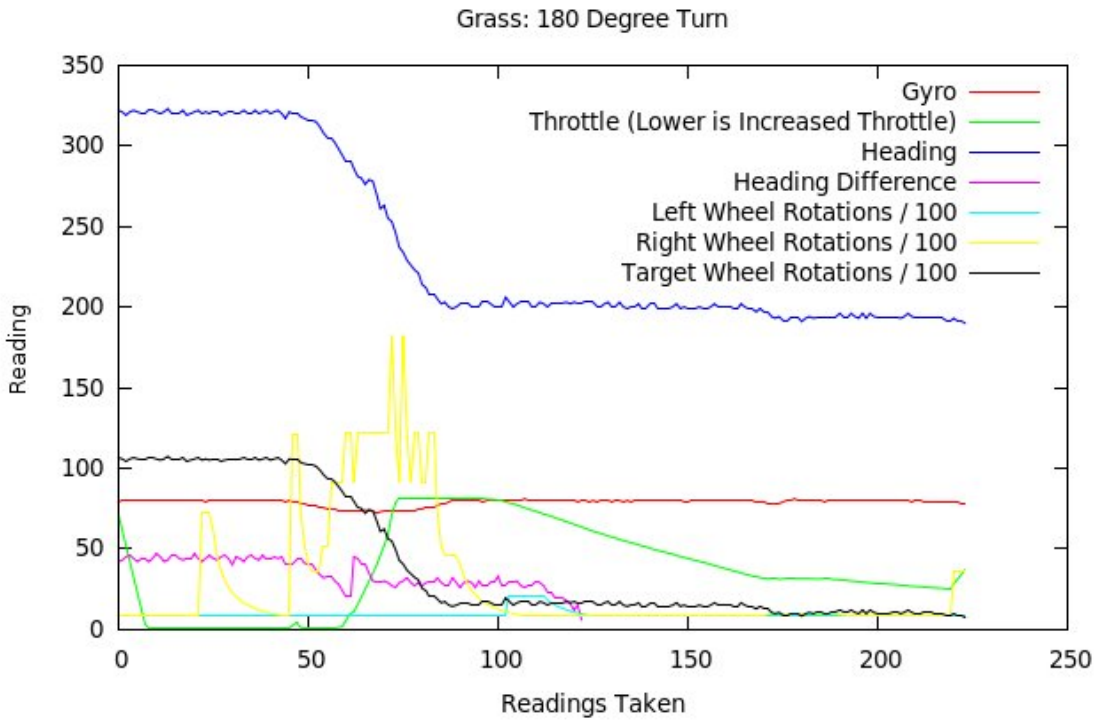


Figure 12.6: A Typical 180 Degree Turn on Grass

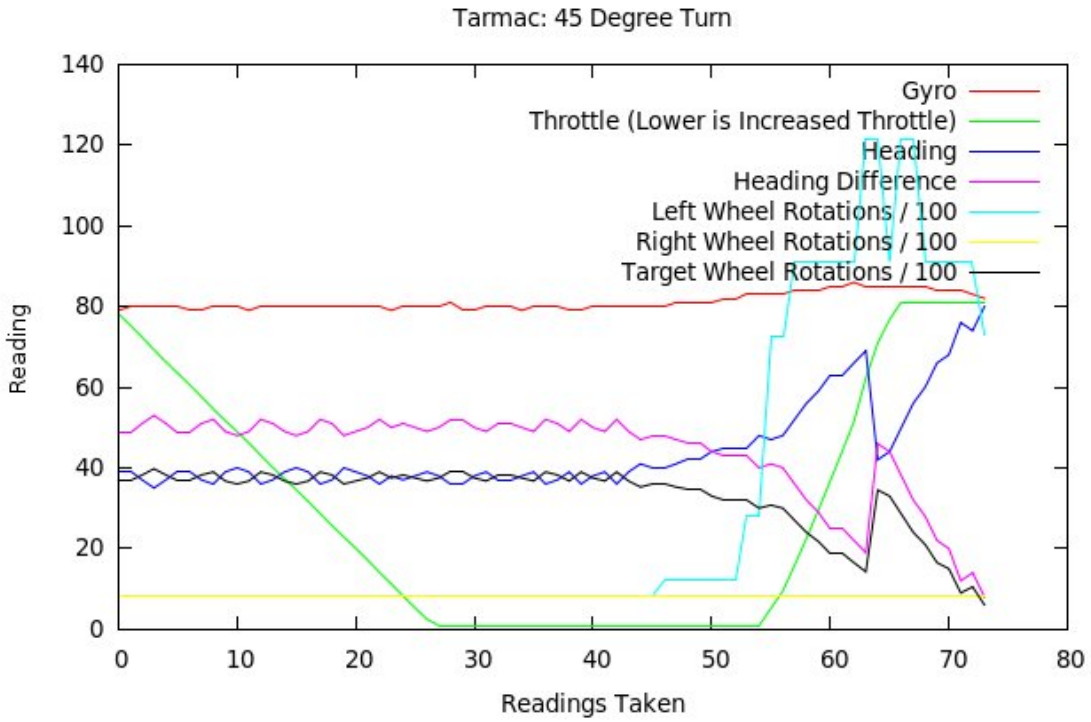


Figure 12.7: A Typical 45 Degree Turn on Tarmac

as with the rest of the data set. This once again is similar to gravel. The reason for this strange data is due to the hill. At this point in time the ARGO was actually turning uphill and this was having an effect on the performance.

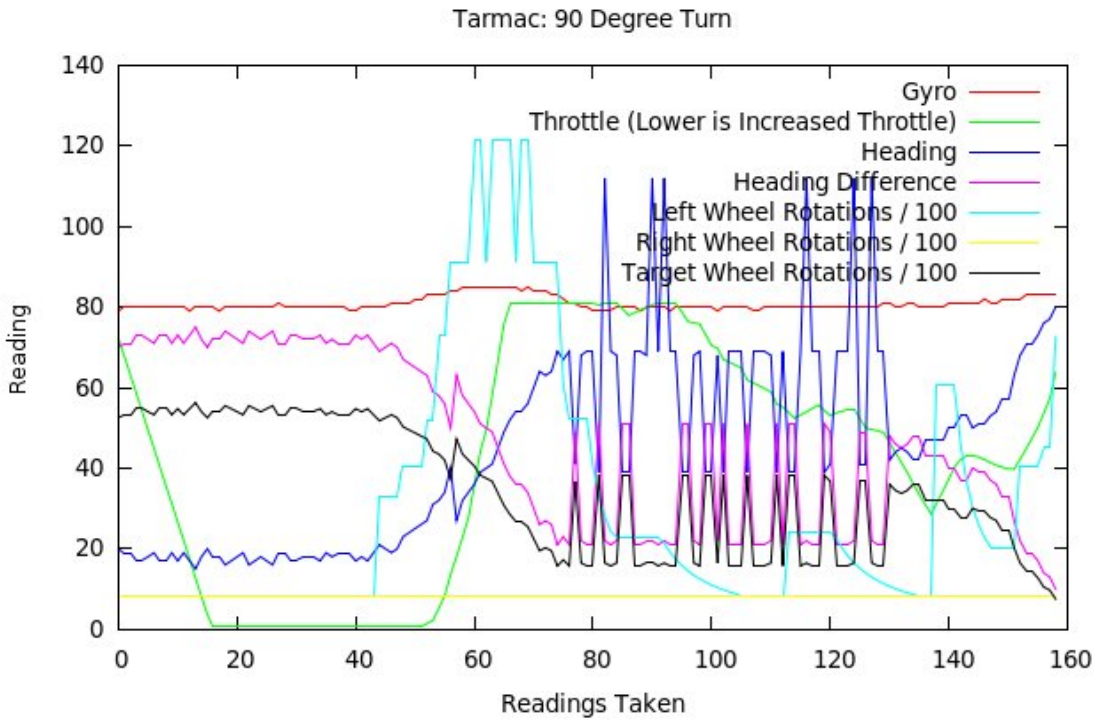


Figure 12.8: A Typical 90 Degree Turn on Tarmac

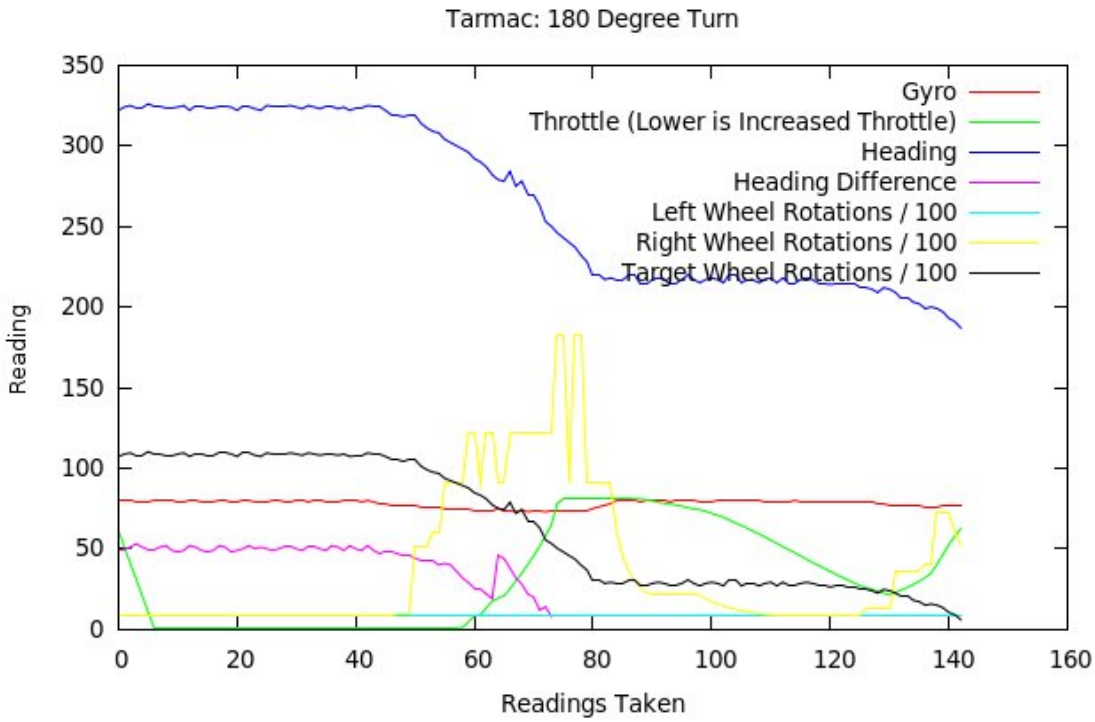


Figure 12.9: A Typical 180 Degree Turn on Tarmac

12.3 Conclusions

It is quite clear from the data gathered that there are significant differences between the different surfaces, particularly between grass/tarmac and gravel. This suggests that it would be possible to detect the surface characteristics and adjust a control system appropriately. Indeed, it may be that a fuzzy logic system would be one method of achieving this where conditions statements such as 'mostly looks like grass', 'is tarmac', 'does not looks like gravel', and 'does not looks like water' are evaluated to make appropriate control decisions.

However, it is also clear that there are environmental effects that change the data. For example, the results for the tarmac data are clearly compromised by the slope. This lead to parts of the data having characteristics similar to gravel.

The question that should be answered is not, "is is possible to identify the surface, for example grass, gravel, tarmac etc?", but "is it possible to identify how the surface reacts?" For example, does a surface react in a similar enough way as grass to be treated as grass would be? Given the data analysed, it would seem that a control system developed specifically for grass would, for example, work with similar precision and efficiency on tarmac. Therefore those two surfaces can be characterised as 'the same' as far as a control system would be concerned whilst gravel would clearly need a different control system for optimum performance.

Chapter 13

Critical Evaluation

"A friendship comes when silence between two people is comfortable."

David Tyson Gentry

13.1 Research Goals

Four research questions were suggested at the start of this project:

1. Can a vehicle such as an ARGO be automated, and what complexity of both software and electronic components are required?
2. What level of accuracy can be expected from a skid-steering system, and to what extent is a control system more or less accurate than a human operator?
3. What effects do surface type have on the operation of the vehicle?
4. Given observed characteristics of the vehicle is it possible to identify the surface type?

With all development and testing work complete a significant portion of time at the end of the project was spent towards answering these questions.

Question 1: Can a vehicle such as an ARGO be automated?

The answer to this question is quite clearly yes. Whilst there have been many challenges during this project, specifically relating to the quality of the hardware components used, it has been shown without doubt that the control of such a vehicle is possible.

Further, the final control system used, whilst relatively primitive, has been shown to have precise, accurate and repeatable control of the ARGO. With further work the accuracy of the control system could be significantly increased. Some future possibilities for the enhancements of the control system include:

- The use of a more expensive compass able to dampen the vibrations that the ARGO is subjected to. This would have the most significant impact on the performance. Most of the issues encountered during development and testing related to issues with the compass.

- The use of surface type detection discussed in the data analysis chapter, perhaps using fuzzy logic systems, to change and adapt the control system according to the surface characteristics could help towards improved accuracy and efficiency on different surfaces. For example, on surfaces such as gravel the target wheel rotations could be adjusted to attempt to reduce the amount of fluctuations seen in the throttle.
- Visual navigation techniques currently under investigation at Aberystwyth [8] could be used and may show significant improvements over the GPS navigation methods. Such a method would be less prone to vibrations or environmental conditions affecting compass readings and GPS coverage.
- The addition of actuators to control the gear leaver. This would then allow the ARGO to reverse and change from low range to high-range when necessary. At this stage, such operations were performed manually via a human operator.

Whilst these changes could improve the efficiency, control, maneuverability and accuracy of the ARGO it is clear that the current control system is adequate for many tasks. It is capable of working on various different surfaces and has been successful in primitive GPS navigation. As such, given the current state of the project, it would be perfectly possible to attach scanning equipment and provide a data file with GPS coordinates such that the ARGO could perform automated scans of fields, rivers etc. In this respect, not only is the control of an ARGO possible, but it has also been achieved with very inexpensive hardware.

Question 2: What level of accuracy can be expected?

Although the accuracy of the ARGO has not been directly discussed, test data (see Appendix C) has been collected and shows that with a primitive control system 75.6% of turns were within 10 degrees of the final target heading. The remaining turns were never more than 5 or 6 degrees from the final destination. This suggests that a reasonable degree of accuracy, and indeed sufficient for a control system, has already been achieved with the current control system.

However, at the moment it is evident that human driven experiments were capable of achieving better accuracy. As humans we are able to both turn the handlebars of the ARGO faster and apply the breaks - removing issues relating to response times that the control system has to deal with.

Despite this, with minor modifications such as the addition of actuators to control breaking and possibly independent control of the left and right disc breaks rather than manipulation of the handlebars could conceivably result in a significant rise in accuracy. In addition, the use of more expensive hardware such as a tilt-compensated marine compass, capable of dampening vibrations and dealing with inclines, would help to further improve the accuracy of the existing control system.

Question 3: What effect does surface type have on the vehicle?

Surface type has been shown to have a significant effect on the performance of the ARGO control system. Areas with loose ground and rocks such as gravel resulted in a significant increase in wheel rotations whilst slower turn rates are achieved. Other surfaces, such as grass or tarmac, have a resistive effect slowing down the increase in wheel rotations. However, once sufficient wheel rotations are achieved the response is quite dramatic with fast turn rates.

Further other surface characteristics such as hills and slopes have a direct impact on the control system and vehicle operations. Whilst traveling up hill throttle values increase, turn rates reduce and wheel rotations increase. The result is interesting data where, for example, tarmac takes on many of the characteristics of gravel. By the same token, when traveling down hill surfaces such as gravel react much more like tarmac and grass - slow buildup of throttle values, followed by fast turn rates.

However, even considering these changes, the effects on the accuracy of the control system are limited. The control system was actually quite good at adjusting its performance to ensure the desired turn angles were indeed reached. The major factor affecting the accuracy of the vehicle (at least on the surfaces tested) was not the surface type but was the noisy sensor data.

Question 4: Is it possible to identify surface type?

As identified during the data analysis chapter it is not directly possible to identify the specific surface type. For example, it does not make sense with regards to the data or control systems to make statements such as 'this is grass'. It was shown that data from grass and tarmac were so similar that it is almost impossible to distinguish between the two.

However, a more appropriate question was suggested: "Is it possible to identify surface characteristics?" That is, is it possible to say that the current surface acts 'like' grass, rather than 'is' grass.

This is indeed possible. The data shows considerable differences between gravel and the other surfaces tested: gravel and tarmac. As such, given unknown data sets it would be quite possible to make the statement: 'this data shows the ARGO performing as though it were on grass.' It is further suggested that this is sufficient in order to adjust the control system of the ARGO to improve the performance on different surfaces. If a surface is so similar to another to be indistinguishable, the control system for both those surfaces should also be indistinguishable.

13.2 Software Objectives

A number of primary and secondary software objectives were identified at the beginning of the project. All primary objectives have been completed and this has lead to the ability to answer the research questions posed.

During the development stages numerous problems were identified that caused delays. In particular the compass, and aspects of the control system that relied on the compass, caused the most problems delaying the whole project by two weeks.

However, despite this, time remained at the end of the project in the final sprint for the development of a secondary objective: GPS Navigation. Whilst simplistic in nature the development of this secondary objective helped to show both the accuracy and control of the low level control system specified in the primary objectives, and the usability of the vehicle for future research such as robotic collaboration, laser scanning, visual navigation etc.

Unfortunately, no extra time remained for the development of additional secondary objectives such as enhancements to the simulator. However, this has in no way affected the success of the project - all primary objectives have been completed and all research questions have been answered. Indeed, many of the secondary objectives suggested could

make more than adequate future dissertation or research projects - all being of considerable size and complexity.

13.3 Methodology

The SCRUM methodology has proven to be highly successful in the project. Whilst the author had experience of various methodologies, including agile methodologies such as XP and Feature Driven Development, SCRUM had never previously been used. The use of SCRUM resulted in a fully functional control system with only a few problems during the development. Further, its use resulted in the many advantages:

- It allowed the author to progressively increase his knowledge of the field of robotics. If it were not for the use of a agile methodology it is highly likely that the authors inexperience would have resulted in failure of the project.
- It was flexible enough to deal with the problems that were encountered during the third sprint which could have been fatal to the project if following a plan-driven methodology.
- It was not at the extreme end of the planning spectrum, meaning that there was enough design work necessary to help keep the project in hand and on track whilst allowing for change.

However, as was discussed when deciding the most appropriate methodology it was pointed out that no single design methodology is ever suited to every project in every way, and this was true for this project. Unfortunately, using an agile methodology had its disadvantages during this project:

- Methodologies are usually suited more to group or team development with many aspects only making sense in this context. For example, during this project aspects such as the SCRUM of SCRUMS, and the daily SCRUM were quite simply inappropriate and as such were not used.
- Having no up-front design did mean that during the project it was often necessary to continually change the design as the project progressed in order to incorporate new features. This, whilst adding flexibility, did often mean that some sections of code had to be thrown away or re-written. This was specifically seen with the communications protocol between the Gumstix and PIC. On numerous occasions that code was re-developed in order to incorporate new features such as the counting of wheel rotations etc.

Ultimately, whilst the SCRUM methodology proved to be successful for this project, if the author were asked to produce software of a similar nature again (with knowledge of the experiences gained during this project) a more plan-driven methodology would definitely be chosen in order to better ensure the real-time constraints and safety-critical aspects of the project. However, if in identical circumstances as this project, where little experience was held, the author would seriously consider the use of SCRUM or a similar methodology such as Feature Driven Development.

13.4 Conclusion

This project has been highly successful with all project goals being met. Whilst over schedule by four weeks, additional secondary features were included such as GPS navigation showing some of the more advanced capabilities of the control system and ARGO.

The project has opened up many possibilities for future research, and numerous suggestions for future enhancements have been put forward to improve the ARGO performance, and accuracy.

The chosen methodology helped the author considerably given his lack of previous experience in the field, yet was perhaps not the most appropriate for a robotics project where there are often safety-critical or real-time constraints that are better met through the use of more plan-driven approaches.

Bibliography

[1] Scott W. Ambler. Feature Driven Development and Agile Modeling, Last Accessed: 20th October 2009. `http://www.agilemodeling.com/essays/fdd.htm`.

[2] Kent Beck. *eXtreme Programming eXplained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[3] Barry Boehm. Get ready for agile methods, with care. *Computer*, 35(1):64–69, 2002.

[4] Dr. James Brasington. Terrestrial Laser Scanning in the Earth Sciences. In *Leica Geosystems Annual Conference on TLS*, Birmingham, 2008.

[5] Xavier Cyril, Gilbert Jaar, and Jean St-Pierre. Advanced Space Robotics Simulator for Training and Operations.

[6] Peter DeGrace and Leslie Hulet Stahl. *Wicked problems, righteous solutions*. Prentice Hall, October 1990.

[7] A. El. Hajjaji and S. Bentalba. Fuzzy path tracking control for automatic steering of vehicles. *Robotics and Autonomous Systems*, 43(4):203–213, June 2003.

[8] Frederic Labrosse. Short and long-range visual navigation using warped panoramic images. *Robotics and Autonomous Systems*, 55(9):675–684, 2007.

[9] Y. Li, K. H. Ang, and G. C. Y Chong. PID control system analysis and design. *IEEE Control Systems Magazine*, 26(1):32–41, 2006.

[10] Mark Neal and Jon Timmis. Timidity: A Useful Mechanism for Robot Control? *Informatica*, 27(4):197–204, 2003.

[11] Eimei Oyama, Nak Young Chong, Arvin Agah, Taro Maeda, and Susumu Tachi. Inverse kinematics learning by modular architecture neural networks. In *IEEE International Conference on Robotics and Automation*, pages 1006–1012, 2001.

[12] Evangelos Papadopoulos, Iosif S. Paraskevas, Thaleia Flessa, Kostas Nanos, Georgios Rekleitis, and Ioannis Kontolatis. The NTUA Space Robot Simulator: Design & Results.

[13] L. Pedersen, D. Kortenhamp, D. Wettergreen, and I. Nourbakhsh. A Survey of Space Robotics. 2003.

[14] Rees Scan Project. Research Goals, Last Accessed: 9th October 2009. `http://www.reesscan.org`.

[15] Dr. Winston W. Royce. Managing the development of large software systems. 1970.

[16] C. Sauze. Control software for a sailing robot. Master's thesis, University of Wales, Aberystwyth, 2005.

[17] Ken Schwaber. SCRUM Development Process.

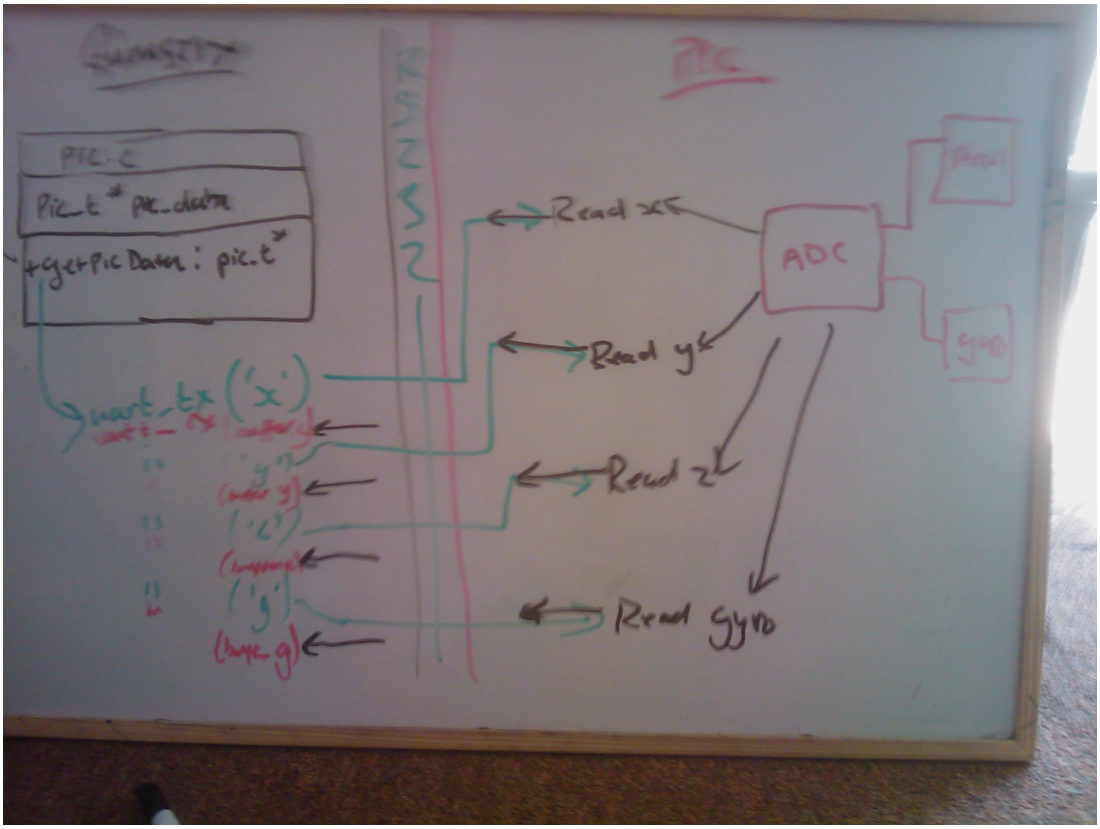
[18] Jeff Sutherland. Agile development: Lessons learned from the first scrum. October 2004.

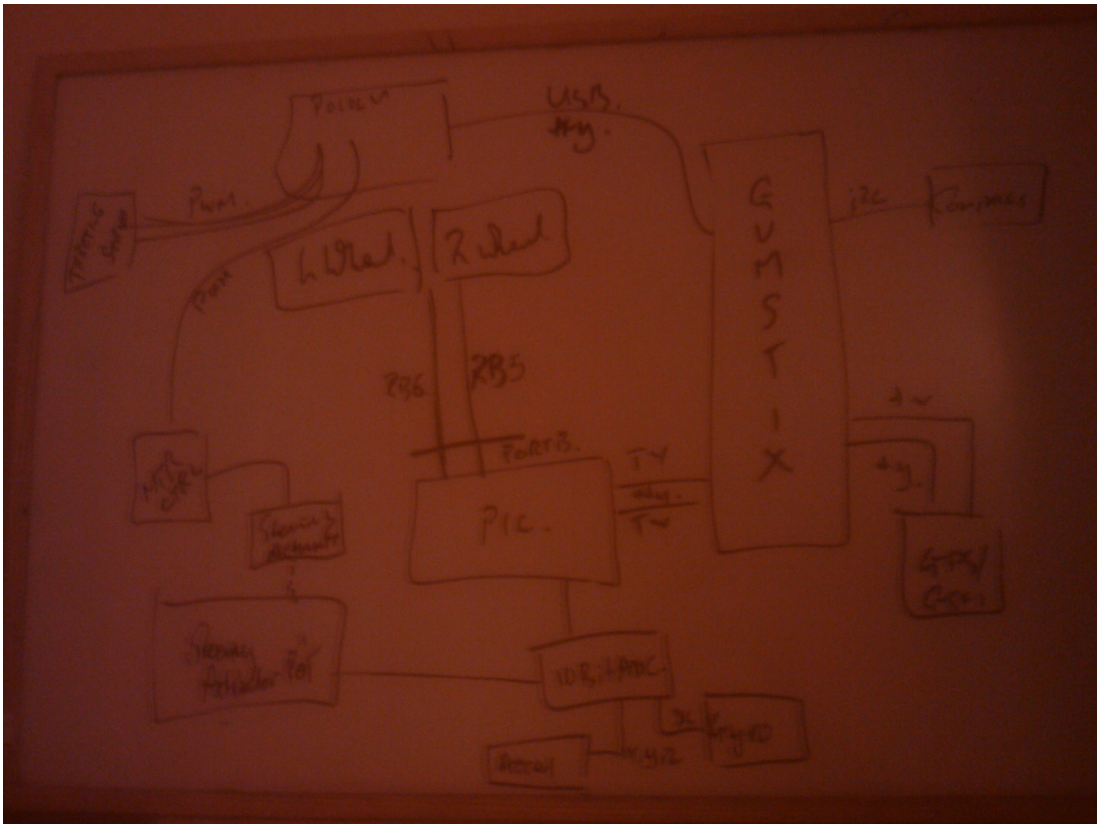
[19] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. January-February 1986.

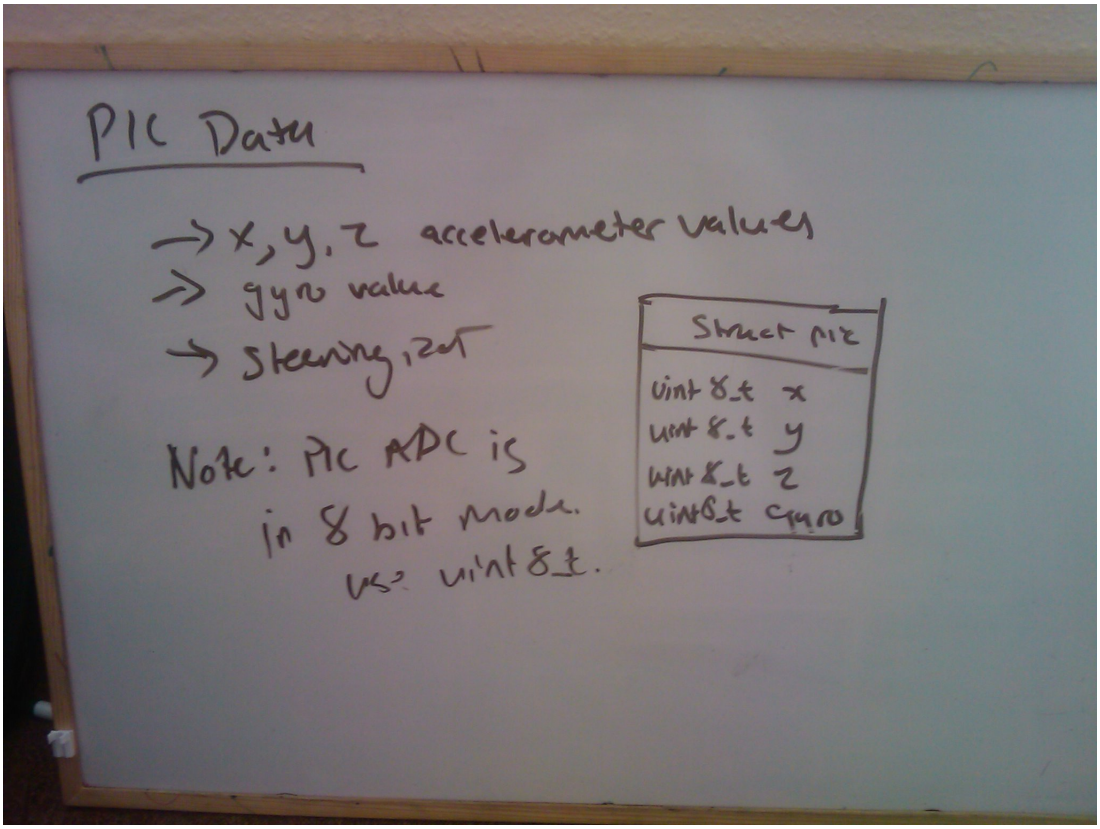
[20] T. W. Vaneck. Fuzzy guidance controller for an autonomous boat. *IEEE Control Systems Magazine*, 17(2):43–51, April 1997.

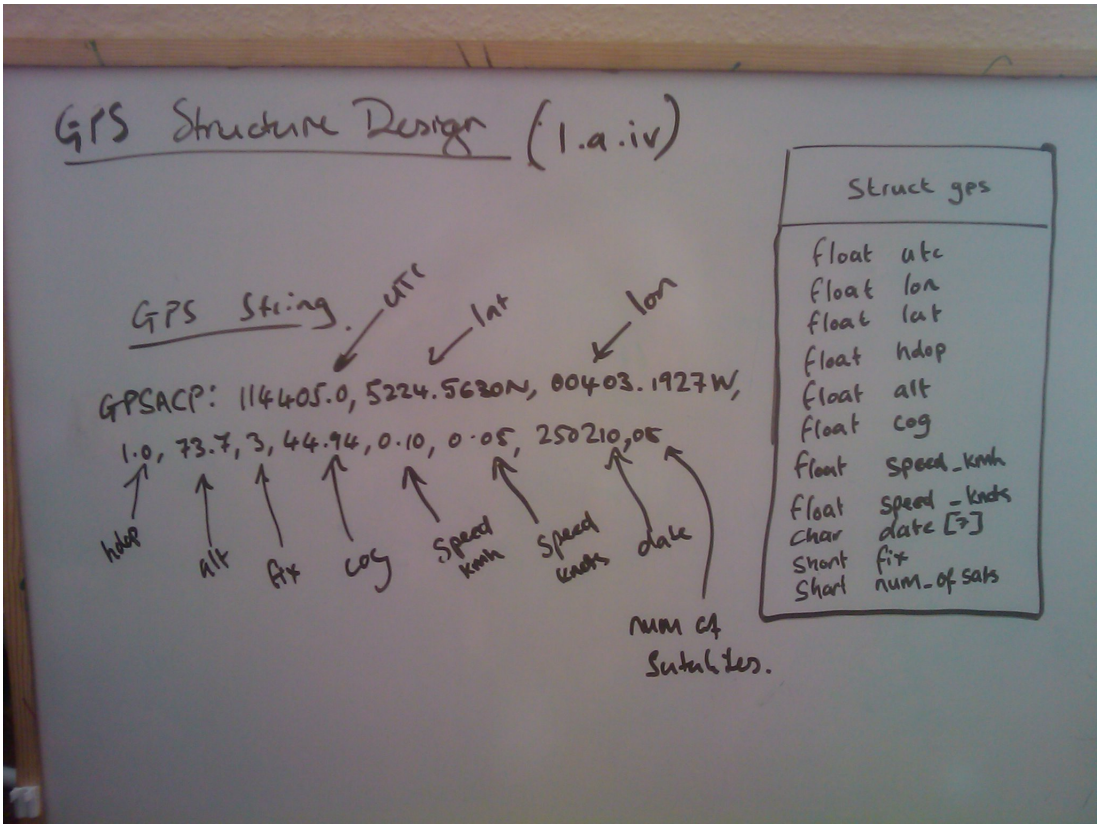
[21] ScienceDaily Washington University in St. Louis. Military Use of Robotics Increases, Last Accessed: 18th October 2009. `http://www.sciencedaily.com/releases/2008/08/080804190711.htm`.

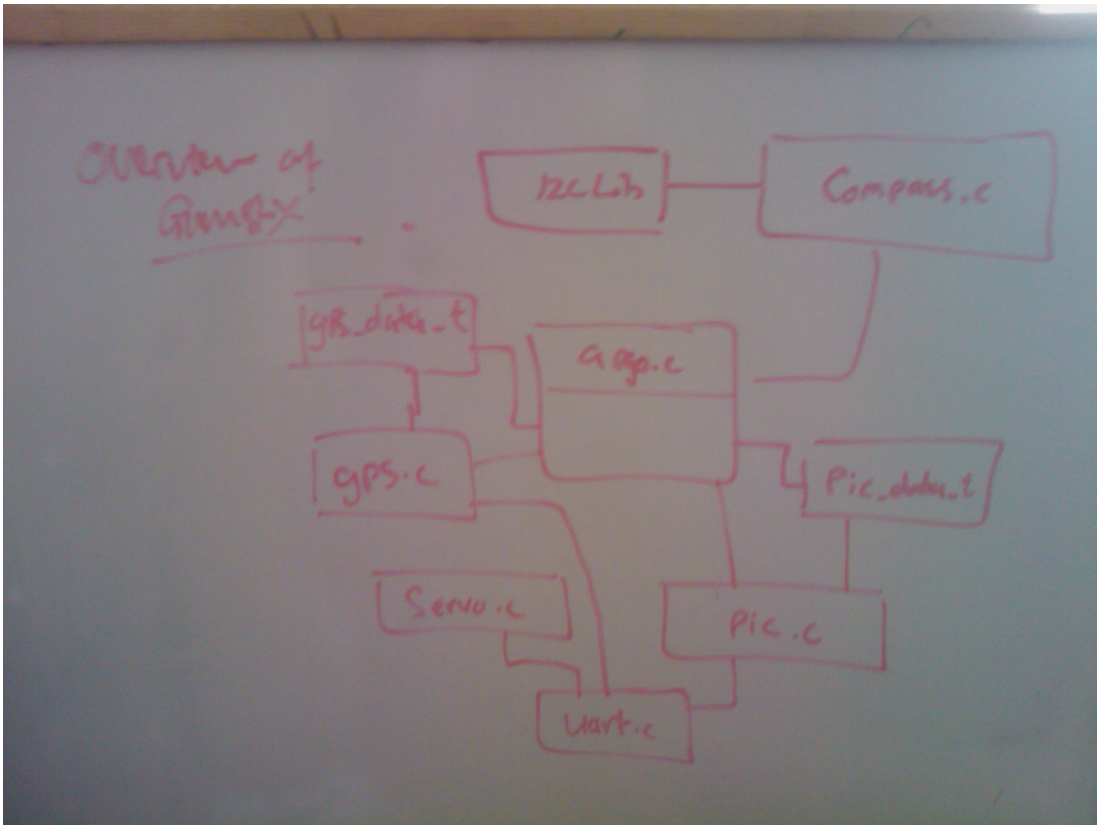
Appendix A
Whiteboard Drawings

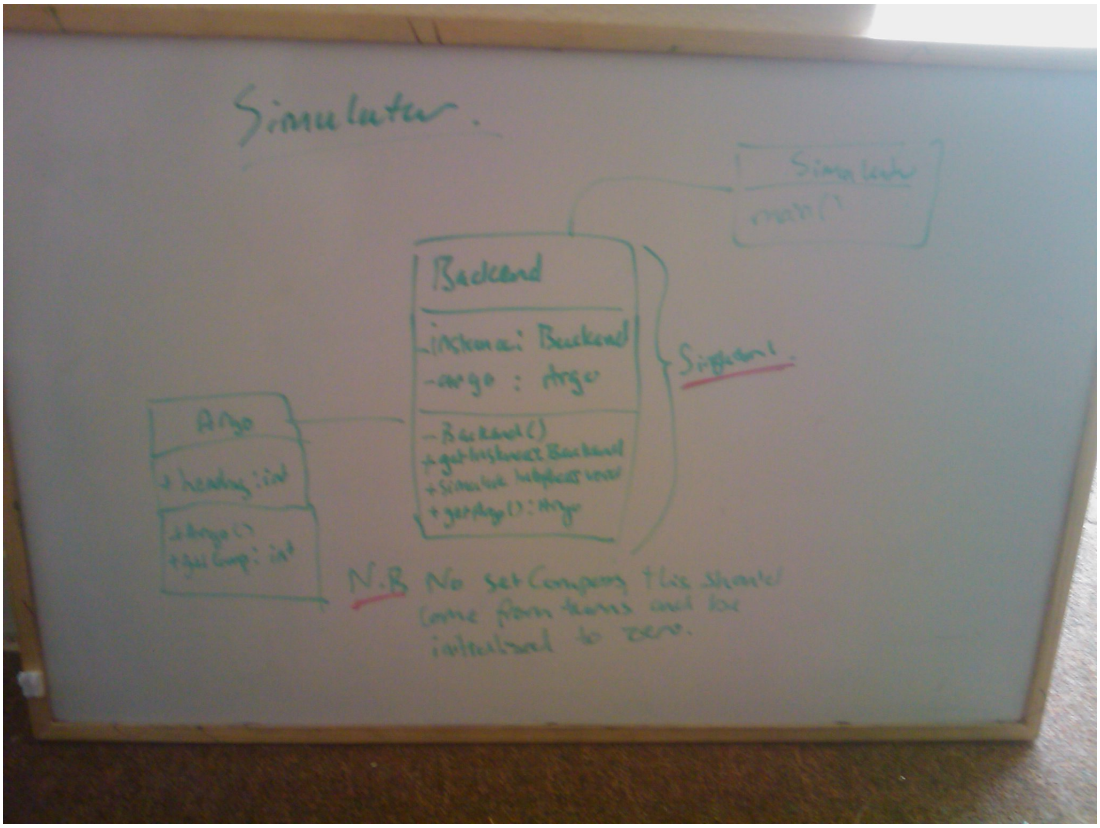

PIC
PIC.C
Pic_t* pic_data
+getPicData : pic_t*
SPI
Read x
Read y
Read z
Read gyro
ADC
uart_tx('x')
('y')
('z')
('g')

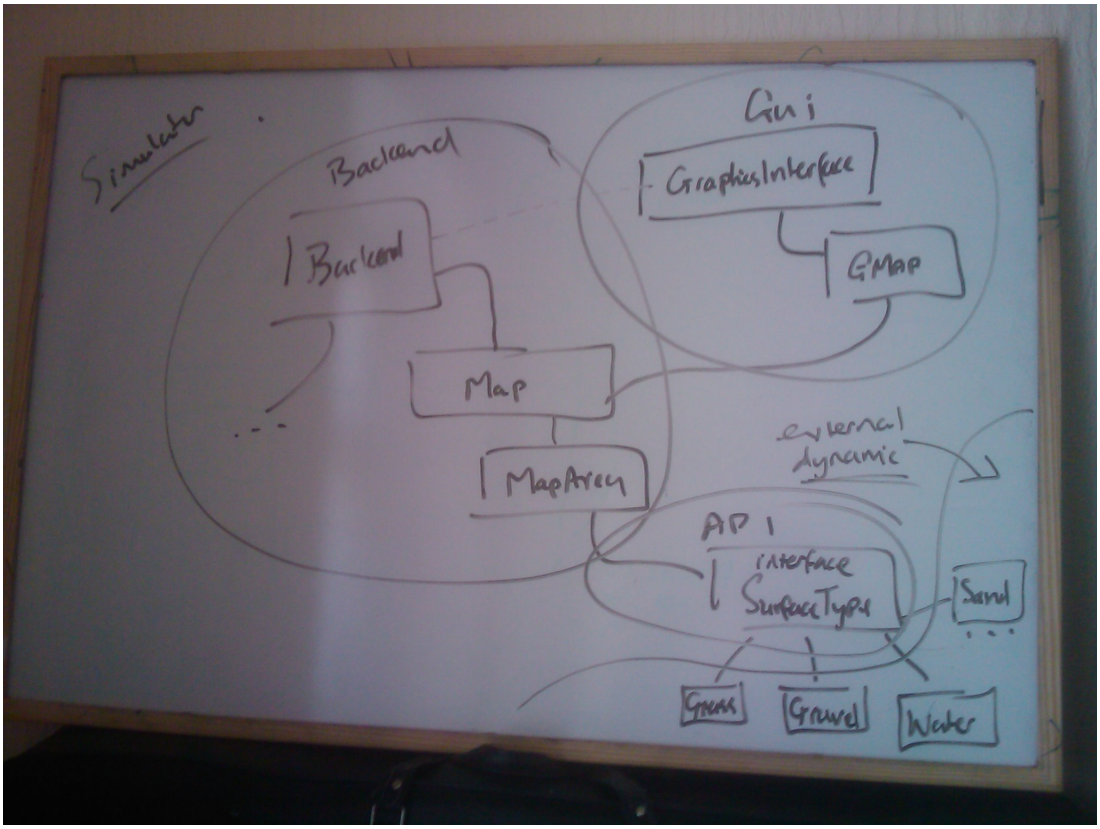

USB
PWM
GUMSTIX
I2C
PIC
SPI
TV

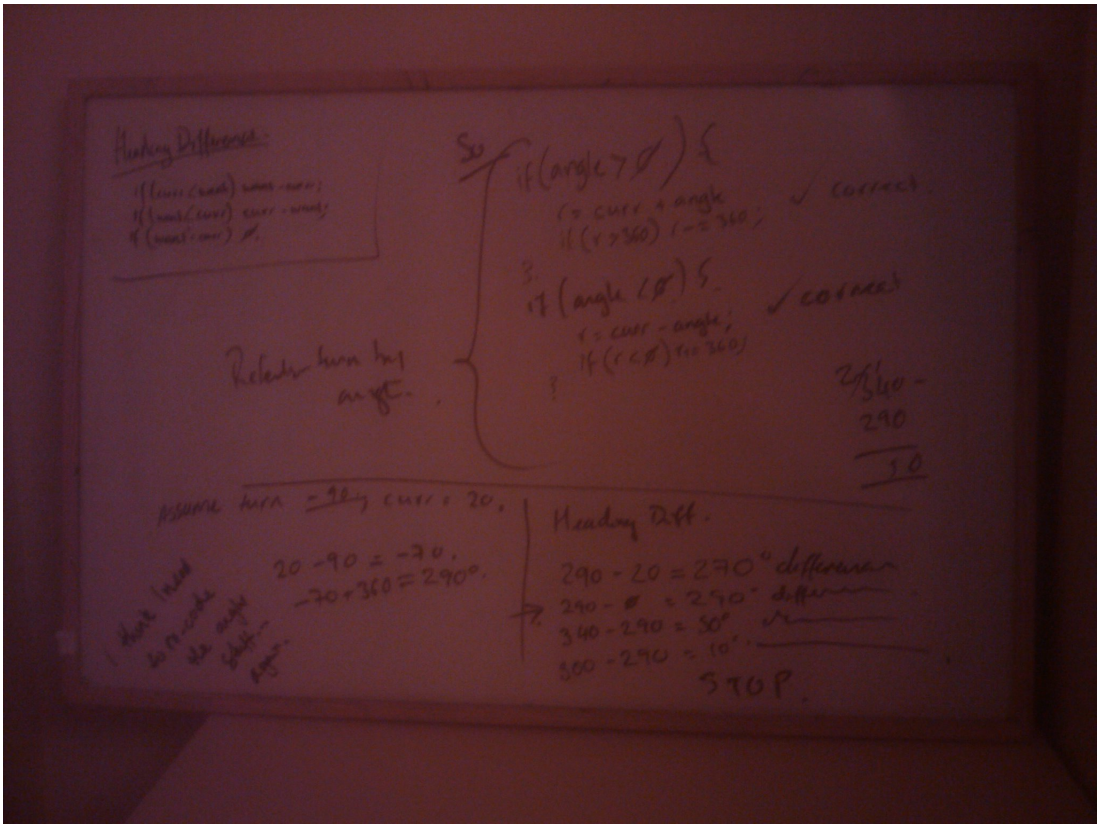

PIC Data
→ x, y, z accelerometer values
→ gyro value
→ steering, zat
Note: PIC ADC is in 8 bit mode. use uint8_t.
Struct pic
uint8_t x
uint8_t y
uint8_t z
uint8_t gyro

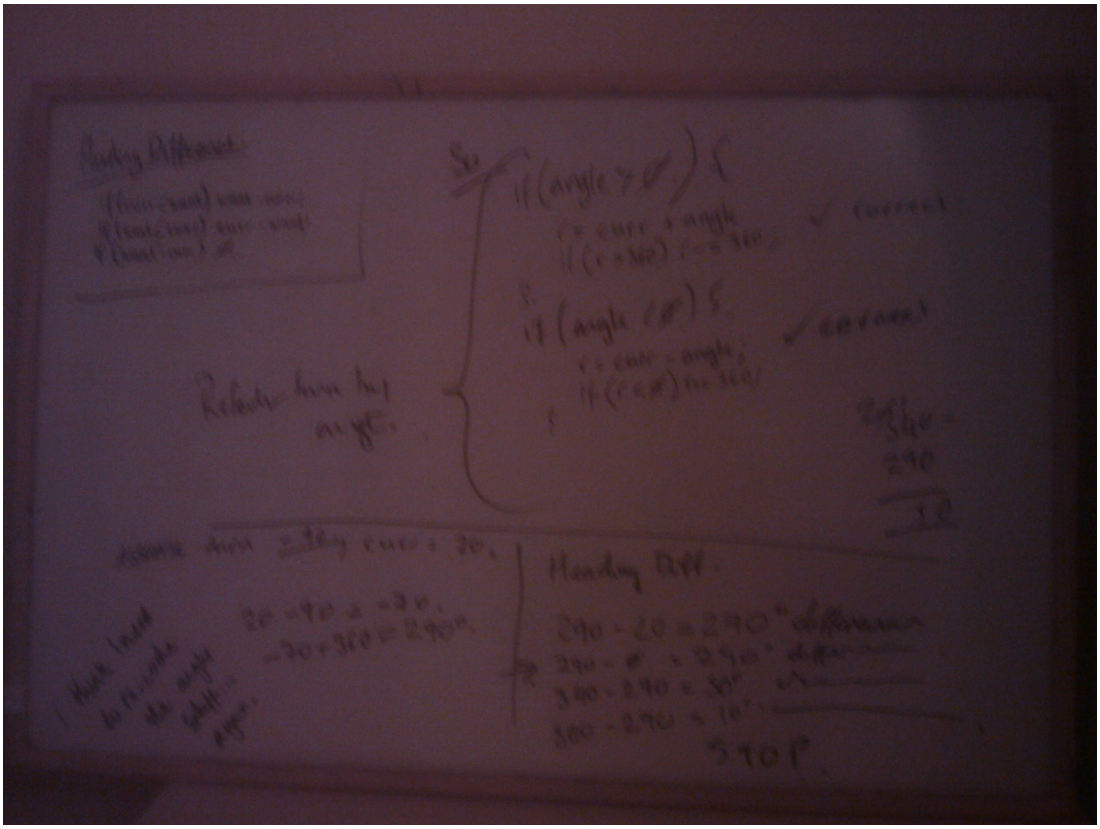

GPS Structure Design (1.a.iv)
GPS String.
utc
lat
lon
GPSACP: 114405.0, 5224.5680N, 00403.1927W,
1.0, 73.7, 3, 44.94, 0.10, 0.05, 250210,05
hdop
alt
fix
cog
Speed kmh
Speed knots
date
num of Satellites.
Struct gps
float utc
float lon
float lat
float hdop
float alt
float cog
float speed_kmh
float speed_knots
char date [7]
short fix
short num_of_sats

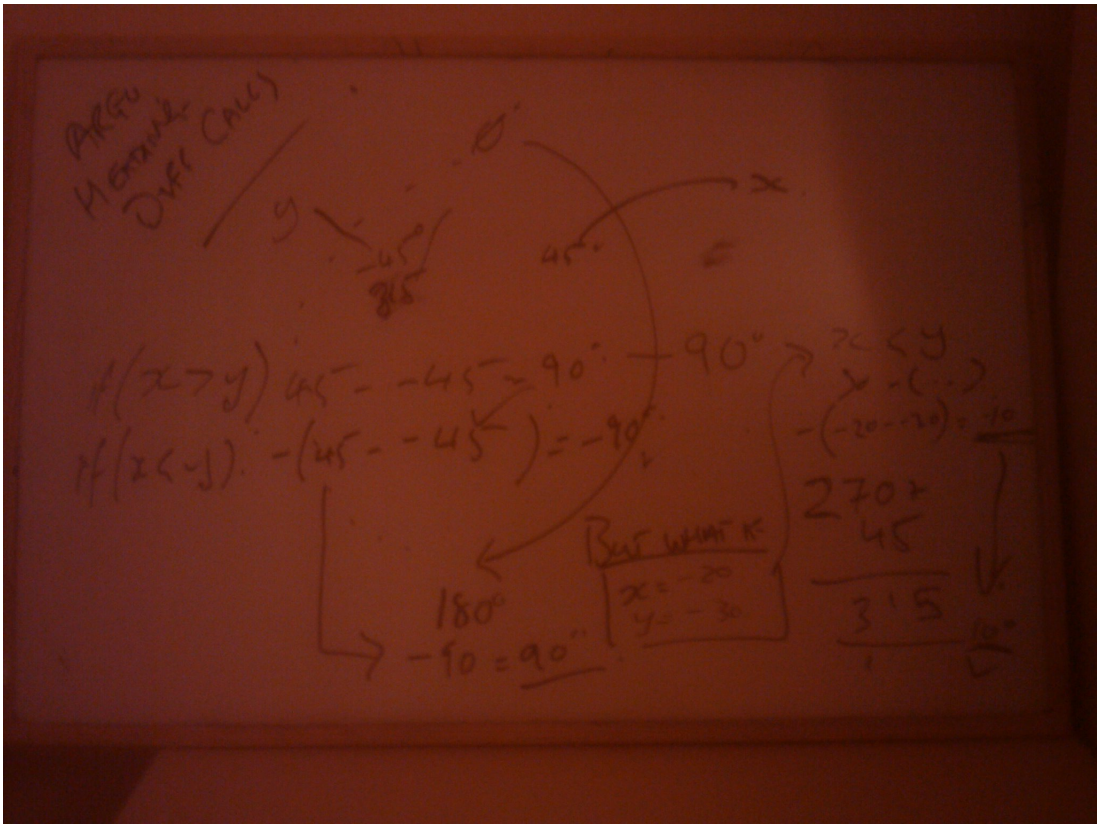

I2C Lib
Compass.c
gps.c
Servo.c
Pic.c
Uart.c

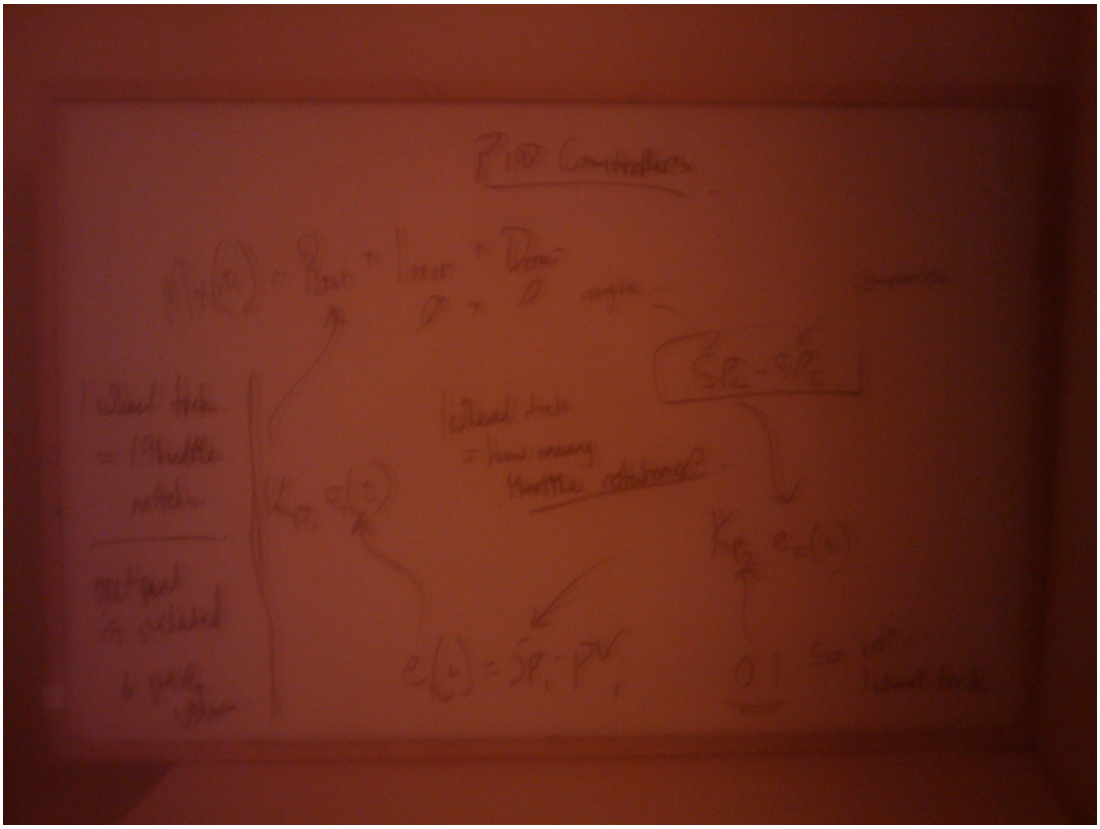

Simulator.
Backend
Args
N.B No setCompass this should come from teams and be initialised to zero.

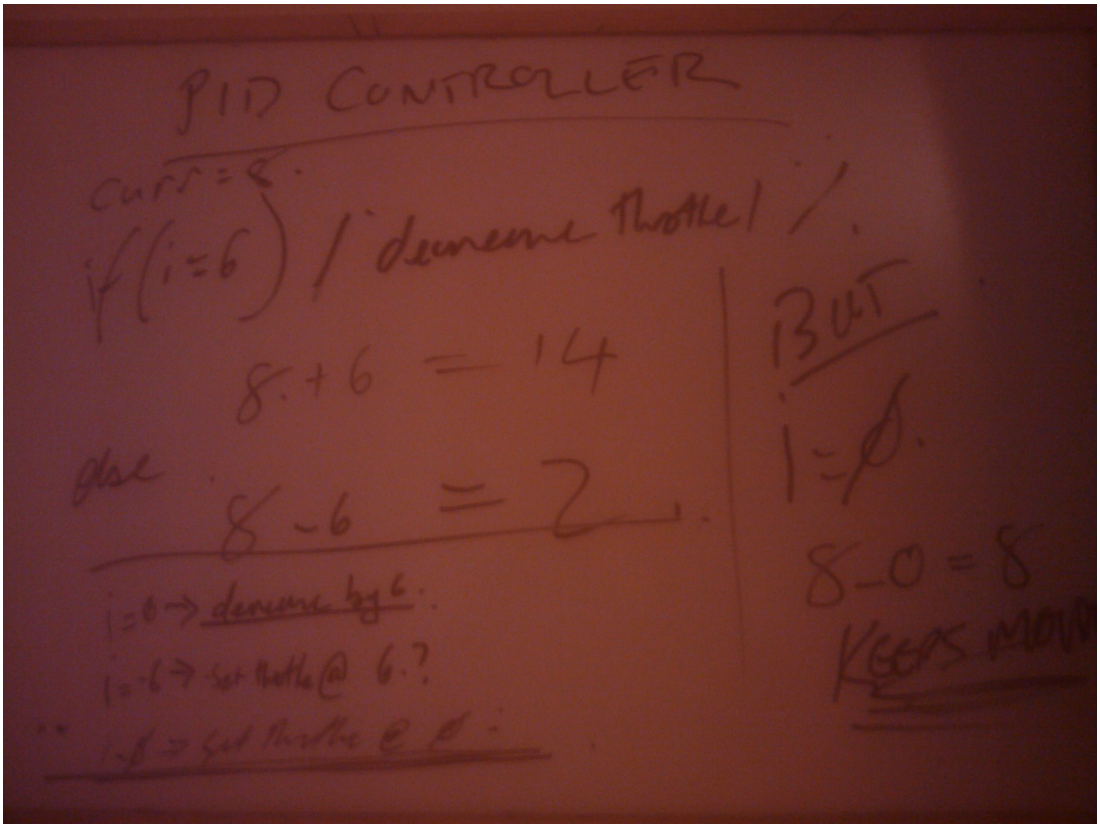

Simulator
Backend
Backend
Map
MapArray
Gui
GraphicsInterface
GMap
external
dynamic
API
interface
SurfaceType
Sand
Grass
Gravel
Water


Heading Difference:
So
if(angle > 0) {
r = curr + angle
if (r > 360) r -= 360;
✓ correct
}
if (angle < 0) {
r = curr - angle;
if (r < 0) r += 360;
}
✓ correct
Relative turn by angle.
360 - 290 = 70
Assume turn = 90, curr = 20.
20 - 90 = -70.
-70 + 360 = 290.
Heading Diff.
290 - 20 = 270° difference
290 - 0 = 290° diff
340 - 290 = 50°
360 - 290 = 70°.
STOP.


Heading Difference:
if (curr - real) < 180:
if (real - curr) < 180:
if (real - curr) == 0:
So
if (angle > 0) {
if (angle < 0) {
Rotate turn by angle.
Heading Diff:
290 - 20 = 270°
STOP


if (x > y) 45 - -45 = 90°
if (x < y) -(45 - -45) = -90°
180
-90 = 90°
270
45
315


PID Controller


PID CONTROLLER
CUTS = 8.
if (i == 6) // decrease throttle //.
8 + 6 = 14
else
8 - 6 = 2.
i = 0 → decrease by 6.
i = 6 → set throttle @ 6.?
i = 8 → set throttle @ 8.
BUT
i = 0.
8 - 0 = 8
KEEPS MOVI

Appendix B
Test Data Sheets

ARGO TEST SHEET

Date	Time	Surface Type
26/3/2010.		GRASS

Weather Conditions

SUNNY, DRY, RAINGING PREVIOUS DAY

Initial Compass Reading

167°.

Test	Expected	Actual/Notes	Pass/Fail
Turn to North	0 degrees +/- 10	9	✓
Turn to East	90 degrees +/- 10, tr	98	✓
Turn to South	180 degrees +/- 10, tr	181	✓
Turn to West	270 degrees +/- 10, tr	271	✓
Turn to North	0 degrees +/- 10, tr	350	✓
Turn to West	270 degrees +/- 10, tl	278	✓
Turn to South	180 degrees +/- 10, tl	179	✓
Turn to East	90 degrees+/- 10, tl	90	✓
Turn to North	0 degrees +/- 10, tl	210 (Compass vibration)	×
Turn to NE	45 degrees +/- 10, tr	41	✓
Turn to East	90 degrees +/- 10, tr	93	✓
Turn to SE	135 degrees +/- 10, tr	148	×
Turn to South	180 degrees +/- 10, tr	167	×
Turn to SW	225 degrees +/- 10, tr	239	×
Turn to South	270 degrees +/- 10, tr	276	✓
Turn to NW	315 degrees +/- 10, tr	326	×
Turn to North	0 degrees +/- 10, tr	151	✓
Turn to NW	315 degrees +/- 10, tl	307	✓
Turn to West	270 degrees +/- 10, tl	269	✓
Turn to SW	225 degrees +/- 10, tl	212	×
Turn to South	180 degrees +/- 10, tl	176	✓

Turn to SE	135 degrees +/- 10, tl	126	✓
Turn to East	90 degrees +/- 10, tl	82	✓
Turn to NE	45 degrees +/- 10, tl	45	✓
Turn to North	0 degrees +/- 10, tl	20 2	X

Note: tl = turning left, tr = turning right.

GPS Tests

Set GPS Points

Point A				Point B			
Lon		Lat		Lon		Lat	
Point C				Point D			
Lon		Lat		Lon		Lat	

All set points are +/- 15m.

Actual GPS Points

Point A				Point B			
Lon		Lat		Lon		Lat	
Point C				Point D			
Lon		Lat		Lon		Lat	

(PASS) / FAIL

Additional Notes

ARGO TEST SHEET

Date	Time	Surface Type
26/3/2010		GRAVEL

Weather Conditions

SUNNY, DRY, RAIN PREVIOUS DAY

Initial Compass Reading

316°

Test	Expected	Actual/Notes	Pass/Fail
Turn to North	0 degrees +/- 10	22	X
Turn to East	90 degrees +/- 10, tr	86	✓
Turn to South	180 degrees +/- 10, tr	179	✓
Turn to West	270 degrees +/- 10, tr	264	✓
Turn to North	0 degrees +/- 10, tr	345	X
Turn to West	270 degrees +/- 10, tl	279	✓
Turn to South	180 degrees +/- 10, tl	190	✓
Turn to East	90 degrees+/- 10, tl	93	✓
Turn to North	0 degrees +/- 10, tl	3	✓
Turn to NE	45 degrees +/- 10, tr	41	✓
Turn to East	90 degrees +/- 10, tr	80	✓
Turn to SE	135 degrees +/- 10, tr	127	✓
Turn to South	180 degrees +/- 10, tr	171	✓
Turn to SW	225 degrees +/- 10, tr	224	✓
Turn to South	270 degrees +/- 10, tr	272	✓
Turn to NW	315 degrees +/- 10, tr	299	X
Turn to North	0 degrees +/- 10, tr	349	X
Turn to NW	315 degrees +/- 10, tl	303	X
Turn to West	270 degrees +/- 10, tl	276	✓
Turn to SW	225 degrees +/- 10, tl	226	✓
Turn to South	180 degrees +/- 10, tl	181	✓

Turn to SE	135 degrees +/- 10, tl	144	✓
Turn to East	90 degrees +/- 10, tl	87	✓
Turn to NE	45 degrees +/- 10, tl	45	✓
Turn to North	0 degrees +/- 10, tl	333.	X

Note: tl = turning left, tr = turning right.

GPS Tests

Set GPS Points

Point A				Point B			
Lon		Lat		Lon		Lat	
Point C				Point D			
Lon		Lat		Lon		Lat	

All set points are +/- 15m.

Actual GPS Points

Point A				Point B			
Lon		Lat		Lon		Lat	
Point C				Point D			
Lon		Lat		Lon		Lat	

PASS / FAIL

Additional Notes

ARGO TEST SHEET

Date	Time	Surface Type
26/3/2010		TARMAC.

Weather Conditions

SUNNY, DRY, RAINING PREVIOUS DAY.

Initial Compass Reading

305°.

Test	Expected	Actual/Notes	Pass/Fail
Turn to North	0 degrees +/- 10	83 *	X
Turn to East	90 degrees +/- 10, tr	90	✓
Turn to South	180 degrees +/- 10, tr	176	✓
Turn to West	270 degrees +/- 10, tr	270	✓
Turn to North	0 degrees +/- 10, tr	351	✓
Turn to West	270 degrees +/- 10, tl	277	✓
Turn to South	180 degrees +/- 10, tl	189	✓
Turn to East	90 degrees+/- 10, tl	100	✓
Turn to North	0 degrees +/- 10, tl	202	X
Turn to NE	45 degrees +/- 10, tr	38	✓
Turn to East	90 degrees +/- 10, tr	82	✓
Turn to SE	135 degrees +/- 10, tr	132	✓
Turn to South	180 degrees +/- 10, tr	192	X
Turn to SW	225 degrees +/- 10, tr	221	✓
Turn to South	270 degrees +/- 10, tr	276	✓
Turn to NW	315 degrees +/- 10, tr	330	X
Turn to North	0 degrees +/- 10, tr	346	X
Turn to NW	315 degrees +/- 10, tl	306	✓
Turn to West	270 degrees +/- 10, tl	262	✓
Turn to SW	225 degrees +/- 10, tl	211	X
Turn to South	180 degrees +/- 10, tl	182	✓

Turn to SE	135 degrees +/- 10, tl	131	✓
Turn to East	90 degrees +/- 10, tl	88	✓
Turn to NE	45 degrees +/- 10, tl	45	✓
Turn to North	0 degrees +/- 10, tl	205 *	✗

Note: tl = turning left, tr = turning right.

GPS Tests

Set GPS Points

Point A				Point B			
Lon		Lat		Lon		Lat	
Point C				Point D			
Lon		Lat		Lon		Lat	

All set points are +/- 15m.

Actual GPS Points

Point A				Point B			
Lon		Lat		Lon		Lat	
Point C				Point D			
Lon		Lat		Lon		Lat	

PASS / FAIL

Additional Notes

Tarmac was very 'hitty', photographs taken.

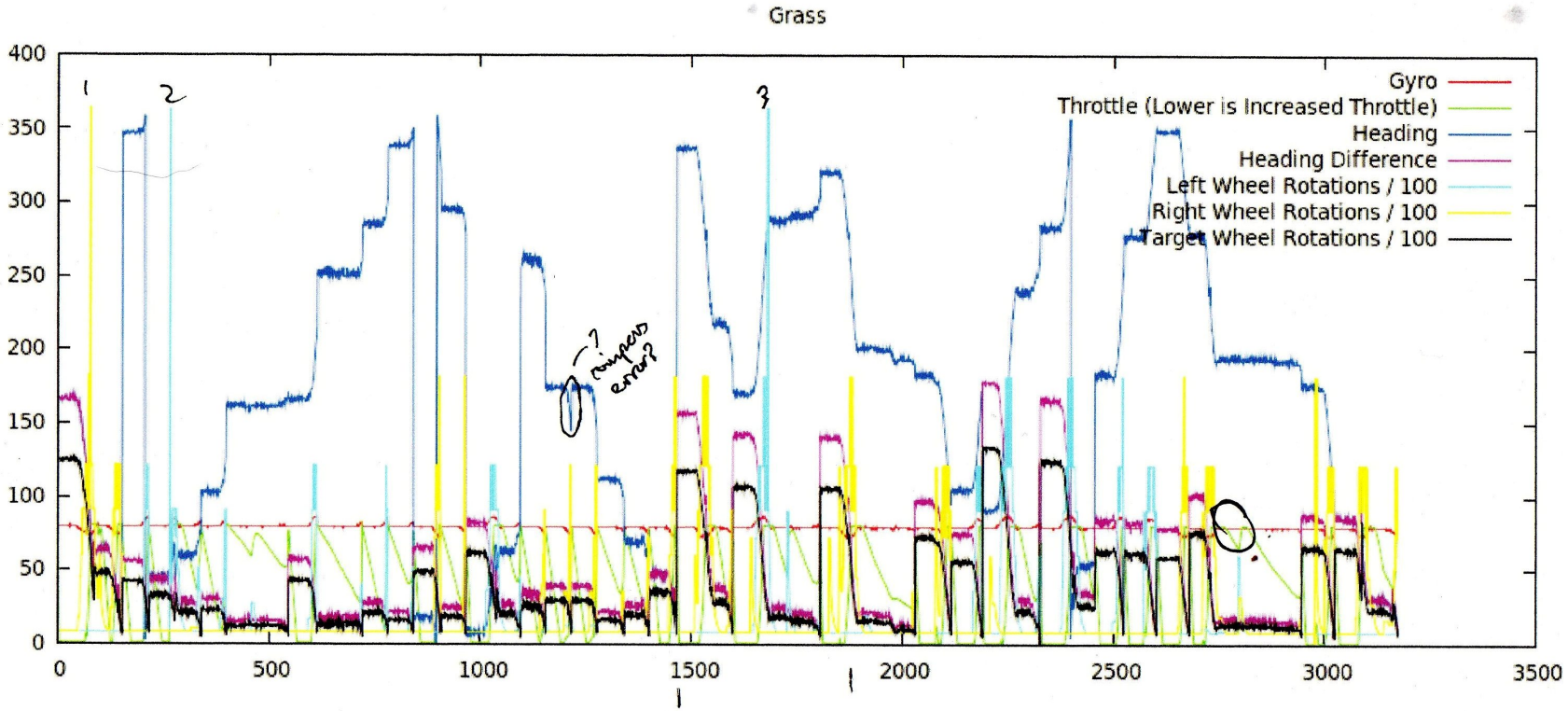
Was on quite a steep slope, this reasons for

1. Ø = 205° and. φ = 83° etc.

↑ see *s.

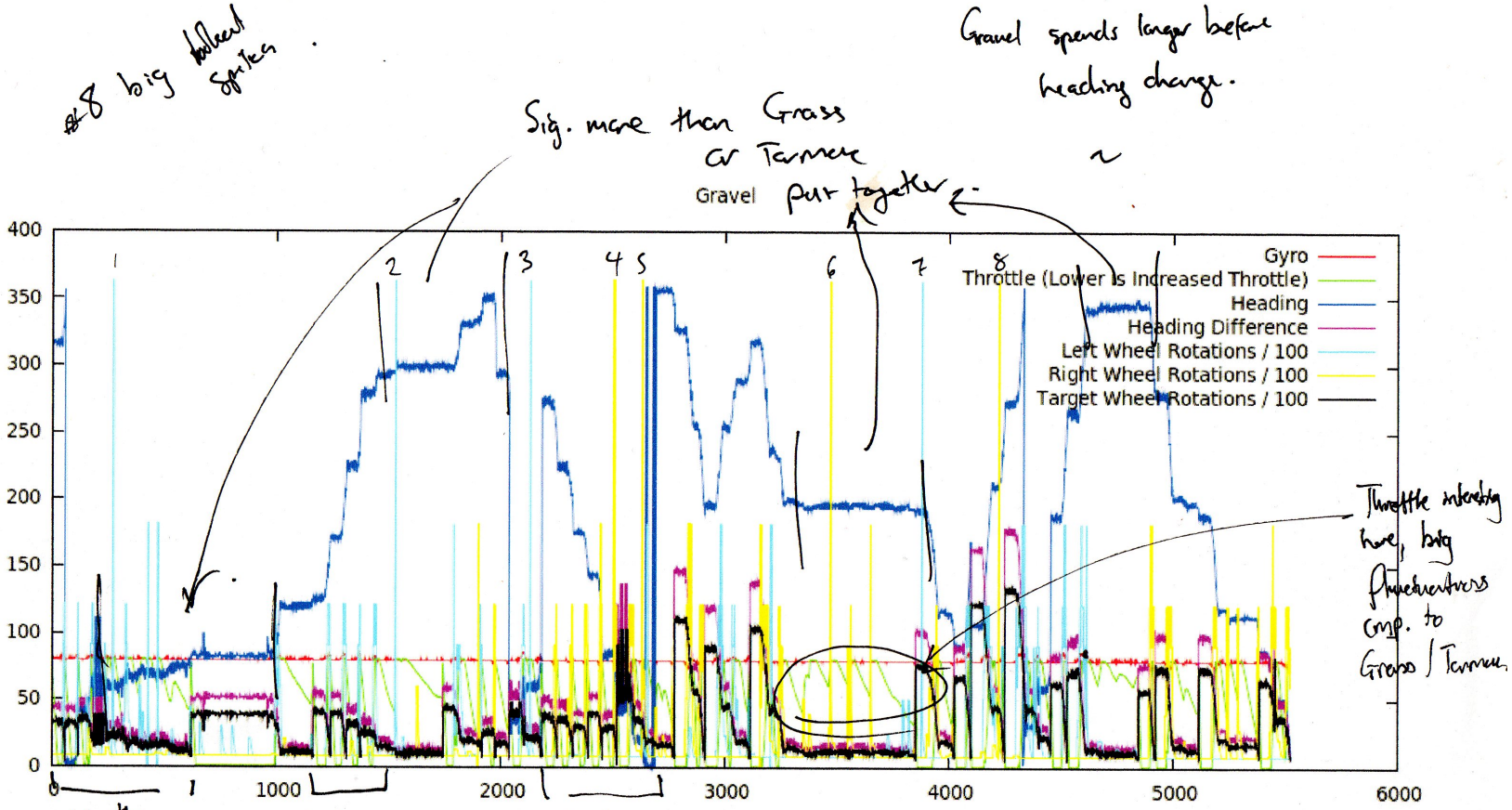
Appendix C
Simple Turn Data Analysis

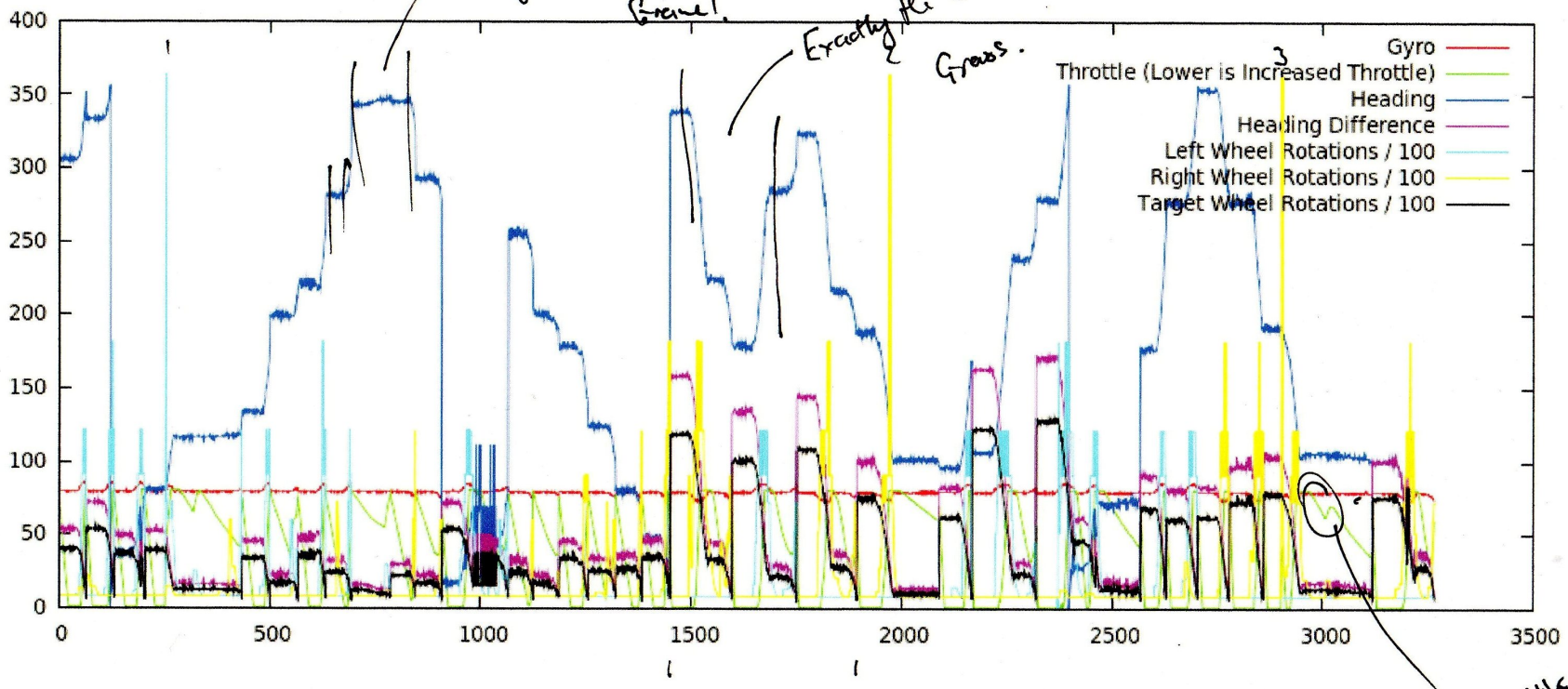
* 3 big steering spikes.



Exactly the same as Tarmac.

Grass shows v. similar to Tarmac with perhaps 2 spikes per turn.


Gravel spends longer before heading change.
Sig. more than Grass or Tarmac
Gravel
Gyro
Throttle (Lower is Increased Throttle)
Heading
Heading Difference
Left Wheel Rotations / 100
Right Wheel Rotations / 100
Target Wheel Rotations / 100
Throttle interesting here, lots fluctuations cmp. to Grass / Tarmac
or this.
light blue.
Don't see this on grass / tarmac.
again with yellow.
What is the max wheel tick overshoot?
What time / position of throttle before wheel rotations?
How long after wheel rotations before change in heading?
How fast is change in heading?
~ 4-6 spikes of wheels



*3 big wheel spikes

more than Grass, significantly less than Gravel.

Exactly the same as Grass.

Exactly the same as Grass.

Same throttle behavior as Grass.

Tarmac shows single, big spike of wheel rotations and massive, fast, heading change.

Appendix D
Example Simulator Files

```
<!--
 SEM9060 Dissertation Project: Development of a Control System and Simulator
 for a Skid Steer Amphibious Vehicle.

 (C) Copyright 2009. Michael F Clarke. All Rights Reserved.
 (C) Copyright 2009. Aberystwyth University. All Rights Reserved.

 The code below is submitted as partial fulfilment of the MEng in Software
 Engineering degree at Aberystwyth University. All code is my own work
 unless otherwise stated.

 This file gives an example of how a world can be developed for the simulator
 environment.
-->
<world>
        <width>800</width>
        <height>600</height>
        <description>
        An sample world which demonstrates the full capabilities of the
        simulator setup.
        </description>
        <area>
                <name>Aberystwyth Beach</name>
                <surface>surfaceTypes.Sand</surface>
                <coordinates>
                        <x>20</x>
                        <y>50</y>
                </coordinates>
                <dimensions>
                        <width>100</width>
                        <height>80</height>
                </dimensions>
        </area>
        <area>
                <name>Gravel Area</name>
                <surface>surfaceTypes.Gravel</surface>
                <coordinates>
                        <x>60</x>
                        <y>100</y>
                </coordinates>
                <dimensions>
                        <width>50</width>
                        <height>100</height>
                </dimensions>
        </area>
</world>
```

```java
package surfaceTypes;

/* SEM9060 Dissertation Project: Development of a Control System and Simulator
 * for a Skid Steer Amphibious Vehicle.
 *
 * (C) Copyright 2009. Michael F Clarke. All Rights Reserved.
 * (C) Copyright 2009. Aberystwyth University. All Rights Reserved.
 *
 * The code below is submitted as partial fulfilment of the MEng in Software
 * Engineering degree at Aberystwyth University. All code is my own work
 * unless otherwise stated.
 */

import java.awt.Color;

import uk.ac.aber.dcs.sem9060.api.SurfaceType;

/**
 * This is the implementation of the Gravel SurfaceType.
 *
 * @author Michael F Clarke (mfc5@aber.ac.uk)
 * @version 0.1
 */
public class Gravel implements SurfaceType {

        /**
         * This is used to get the actual speed of the ARGO.
         *
         * @param throttle The current throttle value.
         *
         * @return The speed of the ARGO.
         */
        public int getSpeed(int throttle) {

                if (throttle < 20) return 12;
                if (throttle < 40) return 3;
                if (throttle < 70) return 1;
                return 0;
        }

        /**
         * Returns the maximum speed that the ARGO can travel at on Gravel.
         *
         * @return The maxmimum speed of the ARGO on gravel.
         */
        public int getMaxSpeed() {
                return 12;
        }

        /**
         * Returns the change in angle at the current throttle value.
         *
         * @return The change in angle to be applied.
         */
```

```
55          public int getAngleChangeValue(int throttle) {
56
57                  if (throttle < 40) return 2;
58                  if (throttle < 20) return 8;
59                  if (throttle < 10) return 15;
60                  if (throttle < 5) return 20;
61                  return 0;
62
63          }
64
65          /**
66           * Returns the minimum throttle value before any movement.
67           *
68           * @return The minimum throttle value before the ARGO will
69           * move on Gravel.
70           */
71          public int getThrottleToGo() {
72                  return 70;
73          }
74
75
76          /**
77           * This method is used to return to the colour that should be used
78           * when displaying the map.
79           *
80           * As this is gravel, the colour is gray.
81           *
82           * @return The color (gray) to be used for displaying areas of this
83           *         surface type.
84           */
85          public Color getColor() {
86                  return Color.gray;
87          }
88
89          /**
90           * This method is used to get the name of this surface type: GRAVEL.
91           *
92           * @return The name of this surface type, specifically GRAVEL.
93           */
94          public String getName() {
95                  return "GRAVEL";
96          }
97
98  }
```

Example GPS data file. Format is first line number of entries. Thereafter longitude, latitude.

```
1 3
2 -4.053083, 52.409367
3 -4.052717, 52.409317
4 -4.052367, 52.409351
```

Appendix E
Resource CD

Included on the CD:

- A soft copy of this report with all LaTeX files and diagrams.
- Copies of many of the papers referenced in this report.
- Various videos and images of the ARGO in action.
- Data files from testing.
- A copy of the SVN repository, including simulator and physical ARGO control source code.
- Test data files and graphs, including GPS test data.